UNIVERSITÉ LIBRE DE BRUXELLES

Faculté des Sciences
Département d'Informatique

Génération de code de systèmes distribués

DEVOS Nicolas

Mémoire présenté en vue
de l'obtention de la licence
en informatique

Sous la direction de M. Thierry MASSART

Année académique 2003-2004

Nous tenons à remercier notre promoteur Thierry Massart, les assistants Cédric Meuter, Alexandre Genon et tout particulièrement Bram de Wachter sans qui ce travail n'aurait pas été possible. Ainsi que tous ceux ayant contribué à l'élaboration de ce mémoire.

Table des matières

Chapitre IV Compilation et distribution

Chapitre V Génération automatique de code LegOS pour Lego-Mindstorms

Chapitre VI Etude de cas : chaîne de montage

Chapitre VII Conclusions

Chapitre I

Introduction

1. Buts et approche suivie

Les objectifs de ce travail sont l'étude et la réalisation, à partir du langage de programmation dSL, d'un générateur de code pour des systèmes de contrôle distribués. Pour réaliser nos objectifs, nous avons recouru à l'utilisation des Lego-Mindstorms comme laboratoire d'expérimentation. Pour cela, nous avons récupéré et adapté un compilateur existant. Nous avons finalement élaboré une étude de cas afin d'évaluer de manière pragmatique le langage dSL et notre compilateur.

2. Les sytèmes réactifs

Un système est une combinaison de parties qui se coordonnent pour former un ensemble. Il peut être vu comme une réunion de composants en interaction : système solaire, système nerveux, etc. Chaque composant a une responsabilité, une fonctionnalité. Un système est un ensemble d'entités telles qu'on ne peut définir la fonctionnalité de l'une indépendamment de celles des autres. L'environnement d'un système est le milieu dans lequel celui-ci évolue.

Un système réactif est un système en interaction permanente avec son environnement. Il réagit aux événements qui se produisent dans cet environnement en réalisant des actions qui influencent éventuellement cet environnement. En d'autres mots, il répond aux stimuli de son environnement. Une représentation d'un tel système est donnée dans la figure 1.1.

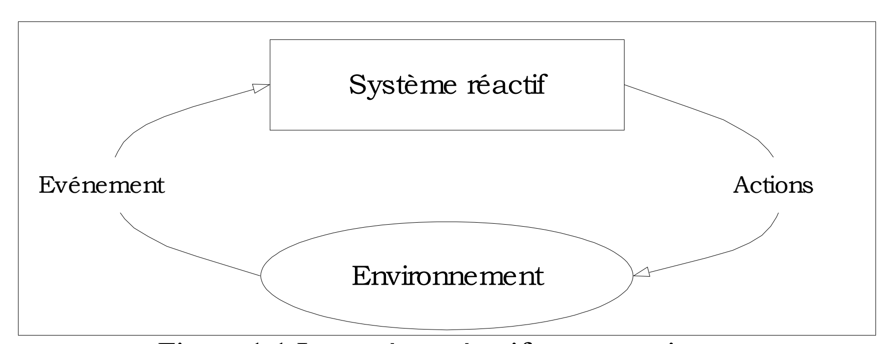


Figure 1.1 Le système réactif et son environnement.

La propriété de réactivité, c'est-à-dire la capacité à réagir en temps réel à des événements, est utile pour modéliser des dispositifs physiques qui répondent à des requêtes par des actions appropriées.

La modélisation de ces systèmes est un domaine aux applications vastes. Ce type de systèmes peut être utilisé pour modéliser le comportement d'un être artificiel par exemple mais aussi de bons nombres de dispositifs de contrôle d'équipements domestiques ou industriels tels qu'un chauffage central ou un système anti-incendie. Lorsque de la fumée est détectée par un des capteurs du système, ce dernier réagit en conséquence, par exemple en déclenchant une alarme.

Il y a différents aspects dans la modélisation de ce type de systèmes. Non seulement parce qu'ils doivent être conçus de manière à réagir correctement et instantanément aux événements mais aussi parce que souvent il s'agit de systèmes distribués. Un système distribué est un système constitué d'un ensemble de composants indépendants et distants, reliés entre eux par un réseau. Différents problèmes se posent alors. D'une part, le bon fonctionnement du système passe par la gestion des communications entre les stations composant le système. D'autre part, il faut résoudre des problèmes liés à la distribution, non seulement en ce qui concerne les données mais aussi le traitement des événements.

Deux approches peuvent être utilisées pour concevoir des systèmes réactifs :
- l'approche synchrone : tous les composants du système évoluent simultanément et effectuent les mêmes actions (par exemple la lecture des entrées ou la mise à jour des périphériques de sortie) en même temps.
- l'approche asynchrone : chaque composant du système évolue de manière autonome indépendemment des autres.

De nombreux langages synchrones comme Esterel [**Ber98, BD91**], Lustre [**CPHP87**] et Signal [**BGL91**] permettent de spécifier le comportement de tels systèmes de manière abstraite et rendent possible leur validation formelle, cruciale dans le cas de systèmes critiques. En pratique, la synchronisation s'effectue par la mise en œuvre d'un site maître qui émet à tous les sites l'ordre d'effectuer une action particulière et attend que tous les sites aient terminé cette action pour passer à la suivante. Faire évoluer de manière synchrone tous les sites peut s'avérer totalement inefficace. De plus, nous ne pouvons imaginer un tel système pour une raison de sûreté. Si un des sites venait pour une raison quelconque à se bloquer ou à être coupé des autres, tout le système serait bloqué. C'est pourquoi l'hypothèse de synchroniser les sites a été rejetée.

3. Structure du mémoire

Dans le chapitre II, le langage dSL est présenté de manière complète ainsi que les problèmes liés à la distribution de code des systèmes distribués.

Le chapitre III contient une brève introduction à la technologie Lego-Mindstorms et à son environnement. En particulier, le langage de programmation ainsi que le protocole de communication utilisés y sont décrits.

Le chapitre IV explique les adaptations effectuées au compilateur dSL existant et décrit en détails la génération de code vers le langage de destination choisi.

Le chapitre V présente une étude de cas comportant la construction et l'implémentation dSL de l'application ainsi qu'une évaluation de la solution.

Les annexes reprennent un ensemble de documentations, de supports techniques, et autres, qui sont en lien direct avec ce travail.

Chapitre II

Le Langage dSL

Dans ce chapitre, nous décrirons brièvement ce qu'est le *Supervision Language*, ses inconvénients et les concepts de son successeur : le *distributed Supervisor Language*. Nous détaillerons les problèmes liés à la distribution de code et la notion de code atomique et séquentiel. Nous terminerons par la définition formelle de la syntaxe et de la sémantique de dSL.

1. Introduction au langage SL

Nous donnerons ici un apercu du *Supervision Language* (SL), ses concepts et ses possibilités ainsi qu'une brève introduction de sa syntaxe.

SL est un langage impératif simple basé sur le langage ST de la norme industrielle IEC1131-3. Il est le prédecésseur du langage dSL. Il a été développé par la société Macq electronique pour modéliser des systèmes industriels distribués et organisés selon le schéma de la figure 2.1 en distinguant trois entités : les contrôleurs programmables (CPs) ou *programmable logic controllers*, un superviseur -qui ensemble constituent le système- et l'environnement. L'environnement et le système interagissent l'un avec l'autre au moyen des dispositifs d'entrée et de sortie des CPs.

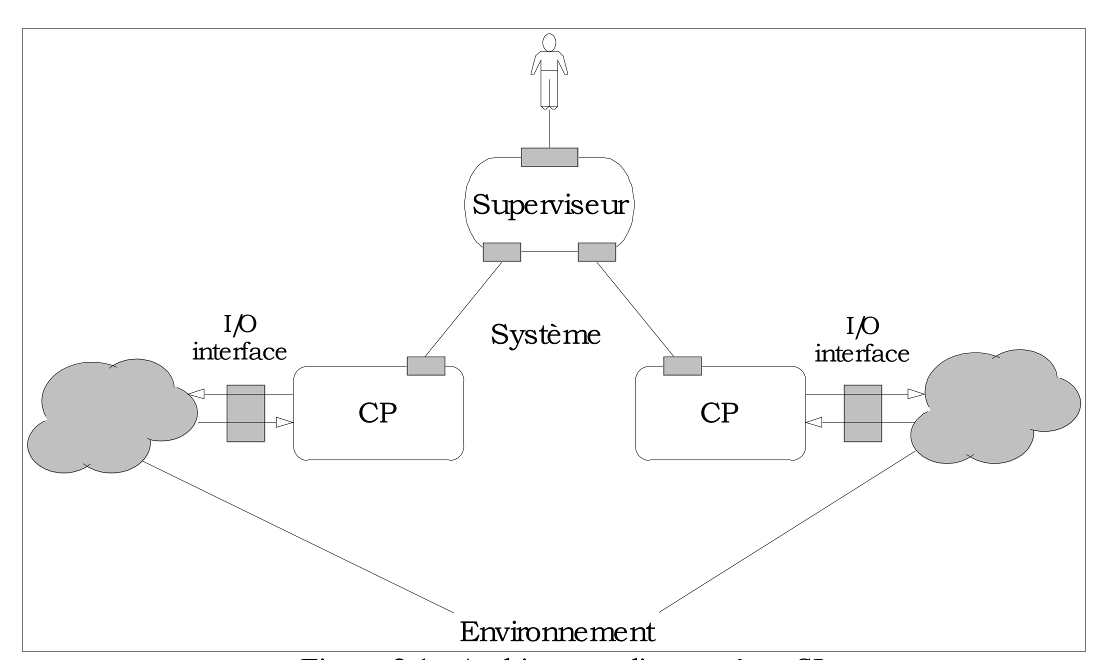


Figure 2.1 : Architecture d'un système SL.

Le superviseur est une unité indépendante possédant deux interfaces. Une vers les différents CPs, l'autre vers l'opérateur. Il effectue le lien entre les différents CPs et l'opérateur.

Les CPs, également appelés automates en raison de leur simplicité, évoluent dans un environnement local et interagissent avec lui par l'intermédiaire de leurs senseurs et actionneurs. L'état de l'environnement local d'un CP correspond aux valeurs de ses entrées, de ses sorties et de ses variables internes. Les CPs ne communiquent qu'avec le superviseur auquel ils transfèrent notamment les valeurs des variables correspondant aux dispositifs d'entrée et de sortie situés sur chacun d'eux. C'est le superviseur qui est chargé de gérer le système en faisant le lien entre les différents contrôleurs. Il leur transmet les valeurs des variables nécessaires et définit leur comportement. Typiquement, le superviseur va mettre à jour certains flags sur les CPs qui seront examinés pendant le traitement.

L'union de tous les environnements locaux des CPs composant le système forme l'environnement global du système. Le superviseur permet ainsi à l'opérateur d'accéder à l'état global du système.

Un CP est responsable de la gestion de ses entrées et sorties ainsi que de l'exécution des tâches qu'il doit effectuer. Le concept traditionnel d'automate induit que chaque CP exécute une boucle, appelée par la suite boucle *Input-Process-Output* divisée en trois phases:

1. *Input* : les différents senseurs d'entrée sont accédés et leurs valeurs sont stockées en mémoire et transmises au superviseur.
2. *Process* : le programme s'exécute en fonction de ses nouveaux inputs et des flags mis à jour par le superviseur. Les résultats sont stockés dans la mémoire du CP et transmis au superviseur.
3. *Output* : les dispositifs de sorties sont mis à jour via la couche hardware.

Les CPs utilisés sont des contrôleurs PIP 4000-1634 dont le langage de programmation est un langage de bas niveau avec un large set d'instructions. Une définition complète de ce langage peut être trouvée dans *Language de programmation du PIP4000-1634* de *Macq* Electronique. En ce qui concerne les périphériques d'entrée/sortie, chaque CP possède plusieurs racks. Un rack est un ensemble de cartes dont chacune possède un certain nombre de connecteurs (ou slot) auxquels sont branchés les dispositifs d'entrée/sortie. Un périphérique est donc défini par trois numéros : un numéro de rack, un numéro de carte et un numéro de slot.

Le langage de programmation du superviseur est SL. Il permet de mettre en oeuvre la gestion des différents CPs composant le système. SL est dit event-driven. Son concept principal est la notion d'événement qui est définie par une condition et un traitement à exécuter. A chaque fois que la condition devient vraie, le traitement est exécuté. SL offre la possibilité de définir différents types de données, des méthodes, et des événements. En outre, d'autres constructions communes à la plupart des langages de programmation telles que les boucles FOR et WHILE sont également disponibles. Beaucoup de fonctionnalités de SL sont reprises dans dSL qui sera décrit plus loin.

2. Inconvénients de SL et motivations de dSL

L'utilisation du langage SL comporte un certain nombre d'inconvénients. Tout d'abord, deux langages différents doivent être utilisés : SL et le langage des CPs. Ensuite, la distribution du code n'est pas automatique. Les communications d'un CP vers le superviseur doivent être spécifiées à la main en langage PIP. De plus, c'est le programmeur qui doit prévoir les transferts de valeur du superviseur vers les CPs. Enfin, il est difficile d'avoir une vision centralisée du système. Le système consiste en un certain nombre de contrôleurs programmables indépendants tous connectés au superviseur.

Afin de pallier ces inconvénients, des solutions classiques (CORBA [**TAN02**], DCOM [**TAN02**], etc.) pourraient être utilisées afin de faciliter le travail du designer. Dans ce cas, l'organisation du système est basée sur le paradigme d'objet. Chaque élément du système est un objet possédant une interface. Dans ces systèmes dits orientés-objet distribué *(distributed object-oriented systems),* des services et des ressources sont disponibles aux clients sous la forme d'objet. Ces solutions se chargent des aspects de communication au travers des interfaces des objets et permettent au programmeur de se concentrer sur les fonctionnalités du système. De plus, l'approche objet distribué offre des facilités dans la conception et la visualistion du système en permettant de représenter celui-ci sous forme d'interactions d'objets. Cependant, ces solutions sont relativement lourdes. De plus, il n'existe pas de sémantique formelle concernant les communications. Le comportement exact du système n'est pas prévisible. Ce qui n'est pas envisageable pour un contrôleur de système critique et qui ne permet pas leur vérification.

C'est pour ces raisons que le langage dSL a été créé. Il reprend la plupart des fonctionnalités de SL mais remédie à ses inconvénients en offrant une sémantique formelle, une vision centralisée du système et une distribution de code transparente et automatique. Dans la suite, nous donnerons une description complète du langage dSL (concepts, syntaxe et sémantique) en décrivant les problèmes liés à la distribution du code et les mécanismes mis en oeuvre.

3. Concepts de dSL

Le langage dSL[**DMM03**] est un langage de programmation orienté-objet qui fournit une distribution de code transparente utilisant des mécanismes de bas niveau. Il permet de concevoir des systèmes de contrôle distribués et permet au concepteur de ne pas devoir gérer explicitement les aspects de distribution du code et de communication entre les contrôleurs du système. Un programme dSL est conçu comme si l'environnement entier est accessible et fournit une vision centralisée d'une application distribuée. Le distributeur dSL se chargera de répartir le code en plusieurs programmes correspondant aux différents sites sur lesquels le système sera distribué.

A l'instar de son prédécesseur SL, dSL est un langage de supervision. La différence entre ces deux langages réside dans la représentation du système. Selon l'approche dSL, le système est constitué d'un ensemble d'automates interconnectés par un réseau et qui évoluent indépendemment les uns des autres. Il n'y a plus de notion de superviseur indépendant. Les communications ne se font plus via le superviseur mais directement entre les contrôleurs.

Comme le montre la figure 2.2, un système implémenté en dSL peut être vu comme une association de deux entités différentes: le système et l'environnement qui interagissent l'un avec l'autre au moyen des senseurs des contrôleurs composant le système. En fonction de l'état de l'environnement -accessible via les senseurs d'entrée et de sortie-, le système effectue les traitements adéquats avec éventuellement des mises à jour des senseurs de sorties. Il est constitué d'un ensemble d'automates ou CPs, appelés par la suite sites, sur lesquels seront répartis les différentes parties du code. Chaque site effectuera une boucle *Input-Process-Output* présentée plus haut. Il prend en charge la gestion des senseurs en accédant régulièrement les senseurs d'entrée et en mettant à jour les senseurs de sortie. La partie *Process* consistera donc à la gestion des communications inter site mais aussi au traitement des événements.

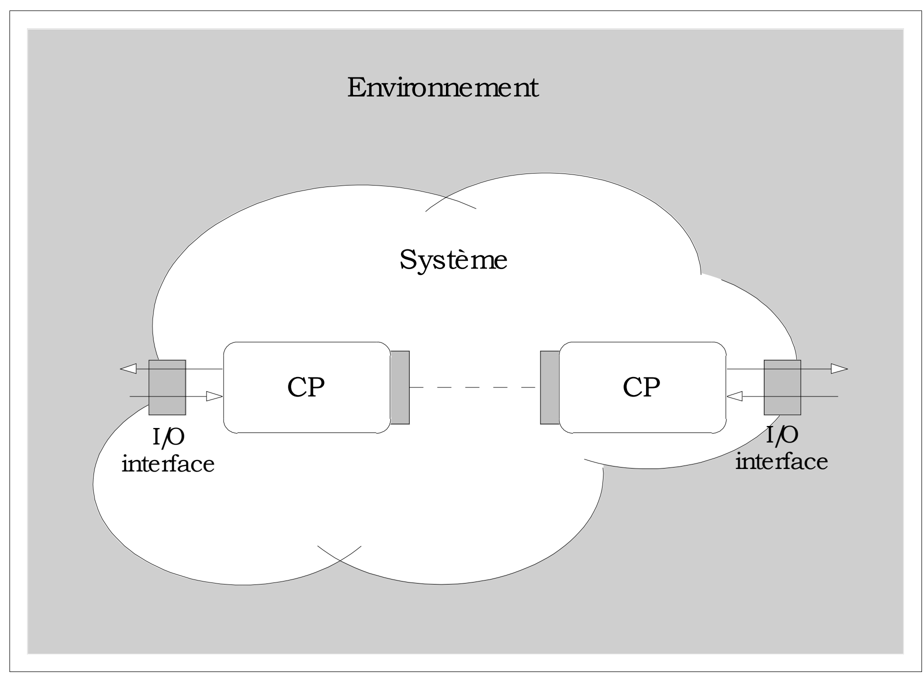


Figure 2.2 : Architecture d'un système dSL.

Remarquons qu'un CP peut être utilisé comme lien entre le programmeur et le système. Dans ce cas, une interface graphique telle que l'interface OBViews développée par Macq Electronique permet de consulter et de modifier la valeur des variables dans le programme.

4. Le problème de la distribution du code

Le code d'un programme dSL est distribué sur les différents sites du système. Deux problèmes se posent :

1. le problème de localisation des variables : de par la nature distribuée du système, une variable peut être utilisée sur plusieurs sites. Le problème de la localisation des variables est de permettre aux sites d'accèder à la valeur d'une variable dont il a besoin.
2. Le problème de localisation des instructions du code : il consiste à prendre en compte la nature distribuée du système et à permettre l'exécution de son code. Les instructions peuvent par exemple être assignée à un site.

4.1. Localisation des variables

Le problème de localisation des variables peut être résolu en utilisant un système à mémoire partagée distribuée (*distributed shared memory* (DSM) [**NL91**]) où l'espace d'adressage est réparti sur les différents sites et la gestion de la mémoire (*memory fault*) est dynamique. Lorsqu'une instruction référence une adresse mémoire qui n'est pas présente dans la mémoire locale du site, une demande de transfert de valeur est broadcastée, c'est-à-dire envoyée à tous les sites. Le propriétaire de la variable demandée lui envoie sa valeur. Même si cette solution offre un environnement distribué transparent, elle souffre de plusieurs désavantages. La gestion d'un tel système est difficile et son efficacité n'est pas garantie. Pour des raisons d'efficacité, certaines données sont dupliquées. Le désavantage principal est que le temps d'attente d'une donnée, c'est-à-dire le temps écoulé entre l'instant de la demande de la donnée et celui de la réponse, n'est pas prévisible. Des problèmes de consistance des données apparaissent alors. En effet, pendant le transfert d'une donnée, il faut empêcher que la donnée soit modifiée sur le site d'origine. Dès lors, pour conserver les exécutions

correctes, le système doit introduire des blocages temporaires de l'exécution. Il en résulte donc des temps d'exécution aléatoires. En outre, le nombre de messages inter sites pourra être important. De plus, la récupération des erreurs et la vérification du code pourront s'avérer délicates.

C'est pour cela qu'une distribution statique a été choisie en dSL. Les variables globales sont réparties statiquement sur les différents sites. Ce qui permet de prévoir à la compilation les communications inter sites qui seront nécessaires au bon fonctionnement du système. Une variable ne peut être distribuée que sur un site et toutes les assignations de cette variable ne pourront être effectuées que sur ce site. La distribution statique repose sur le fait que plutôt que de demander la valeur d'une variable pendant l'exécution, on va s'assurer que la valeur soit disponible et pertinente à tout moment de l'exécution.

4.2. Localisation des instructions de code

Après avoir attribué statiquement les variables à des sites, il faut résoudre le problème d'exécution des instructions. Une solution est la migration de tâches (ou *thread migration*). La détermination de l'endroit où s'exécute une instruction donnée peut se faire de manière dynamique, c'est-à-dire pendant l'exécution. Lorsqu'une tâche en cours d'exécution doit être transférée sur un autre site (par exemple moins chargé), son exécution est arrêtée et son contexte, c'est-à-dire ses variables locales et le program counter, est transféré sur l'autre site où il est restauré et où l'exécution de la tâche se poursuit. En ce qui concerne le code des tâches, il est soit présent sur tous les sites, soit transféré de site en site avec son contexte. Ce type de solution pose évidemment plusieurs problèmes notamment de performance et de surcharge des communications. Transférer tout un contexte coûte en temps et en ressource réseau. Des améliorations de performances sont possibles en particulier lorsque qu'un processus est migré d'une station lourdement chargée vers une station légèrement chargée ou plus puissante. Il existe un certain nombre d'algorithmes pour répartir la charge d'un système distribué (*load-balancing, ...*). Mais en pratique, la distribution est déterminée de manière à limiter les communications. En plus, les systèmes utilisant la migration de tâches sont relativement difficiles à gérer.

Pour ces raisons, la distribution du code se fait également de manière statique dans l'environnement dSL. Une tâche pourra être divisée en différentes parties successives qui seront réparties uniquement sur différents sites en fonction de la répartition des variables globales et de la capacité des sites. Lorsque l'exécution d'une partie de tâche est terminée, la partie nécessaire du contexte de la tâche est transférée sur le site de la partie de code suivante, le contexte est restauré et la première instruction de celle-ci est exécutée. La distribution statique permet donc de simplifier la gestion du code (aucun code n'est dupliqué ou transféré), de limiter au maximum les messages échangés entre les sites (seules les valeurs nécessaires sont transférées) et en outre de connaître la charge exacte de chaque site.

Les communications inter-sites sont donc définies à la compilation de manière statique, automatique et transparente, en fonction de la répartition des variables globales. Nous verrons plus loin comment la répartition est fournie par le programmeur du système dans le code du programme dSL.

5. Event-driven

Comme son prédécesseur, dSL est un langage event-driven utilisé pour modéliser des systèmes nécessitant une réaction immédiate à un événement et son traitement instantané. Le contrôle de processus industriel recquiert en effet ce type de comportement. C'est le principe de causalité (action – réaction) définissant les systèmes réactifs décrits dans le chapitre *I. Introduction*. La conception

d'un système en dSL passe donc par la définition des événements auxquels le système devra réagir. Lorsqu'un événement se produit, en l'occurence ici lorsqu'une condition booléenne définie sur des variables globales du système passe de l'état faux ou inconnue (cf. *9. La syntaxe de dSL*) à l'état vrai, la séquence d'instructions associée à l'événement est exécutée.

6. Code atomique et séquentiel

Le paradigme utilisé dans le monde des systèmes industriels recquiert une réaction immédiate aux événements et leurs traitements instantanés. Le design de dSL est dicté par ce paradigme. Cela implique qu'en pratique, il n'y ait aucune synchronisation qui pourrait retarder l'exécution.

Une distinction entre deux types de code est effectuée dans le cas de programmes écrits en dSL. D'une part, le code atomique ou event-driven exécuté de façon atomique et non distribué. D'autre part, le code non atomique ou séquentiel qui peut être distribué entre les différents sites. Les instructions séquentielles d'un programme dSL peuvent être découpées en partitions réparties sur différents sites.

Par défaut, le code des événements, c'est-à-dire les instructions d'une construction WHEN, et celui des méthodes est considéré comme atomique et doit donc être conservé sur un même site. Toutes les variables apparaissant dans la condition ou dans le traitement de l'événement doivent donc obligatoirement être situées sur le même site. La figure 2.3 montre un exemple de WHEN où les variables `x` et `y` doivent être réparties sur le même site.

```
WHEN x > 0 THEN
      y := 0 ;
END_WHEN
```

Figure 2.3 : Exemple de WHEN.

Cette contrainte, appelée contrainte atomique, limite les possibilités du système. Il a donc été nécessaire de définir un moyen de relâcher cette contrainte. Deux mécanismes ont été mis au point :

1. « LAUNCH » qui permet d'appeler une séquence ou une méthode sur un autre site de manière asynchrone. Il faut remarquer que plusieurs instances d'une séquence ne peuvent être exécutées simultanément.

2. « TILDE » qui permet de mettre en œuvre une copie locale d'une variable distribuée sur un autre site. La copie locale fera référence à la dernière valeur connue d'une variable tildée. Dans certaines circonstances, l'usage de ce tilde peut causer des problèmes si la différence entre la valeur tildée et la vraie valeur porte à conséquence. En pratique, le programmeur devra veiller à minimiser les impacts néfastes que les temps de transfert de valeur pourraient avoir pour que ceux-ci ne mettent pas en danger le bon comportement du système.

Ces deux mécanismes doivent être introduits par le concepteur du programme. Ils ne sont pas générés automatiquement par le compilateur pour des raisons de sûreté et de performance. L'usage du LAUNCH impose une communication inter-site pour ordonner l'exécution d'une méthode ou d'une séquence. L'usage d'un tilde impose des communications inter-site à chaque fois que la variable tildée est assignée pour mettre à jour les différentes copies distantes de la variable.

Soit le code de la figure 2.3 avec les variables x et y situées sur des sites différents. La contrainte atomique empêche de distribuer ce code. La figure 2.4 présente les deux possibilités offertes au programmeur pour relâcher cette contrainte. La première requiert la définition d'une méthode ou d'une séquence exécutant le traitement de l'événement (en l'occurence ici y:=0) et qui sera distribuée sur le site de y. Le WHEN sera traité sur le site de x et enverra un ordre d'exécution au site de distribution de la méthode ou de la séquence (celui de y). La deuxième permet de distribuer le WHEN sur le site de définition de y. A chaque fois que x sera assignée, un message sera envoyé au site de y pour signaler le changement de valeur. La condition sera vérifiée dès le traitement du message. En pratique, lorsqu'une variable est assignée, les WHEN portant sur cette variable sont examinés.

```
WHEN x > 0 THEN
    LAUNCH y_egal_0() ;        (1)
END_WHEN

WHEN ~ x > 0 THEN
    y := 0 ;                   (2)
END_WHEN
```

Figure 2.4 : Relâchement de la contrainte atomique.

7. Avantages de dSL

Le langage dSL offre plusieurs avantages (par rapport à son ancêtre SL notamment). Tout d'abord, les programmes dSL font preuve de flexibilité. Une modification d'un dispositif d'entrée ou de sortie, c'est-à-dire essentiellement la modification de l'emplacement d'un senseur, ne nécessite à priori aucun changement dans le code du programme (sauf vis-à-vis du respect de la contrainte atomique). Les changements dans les primitives d'accès seront générées automatiquement par le compilateur.

Deuxièmement, c'est un langage extrèmement simple avec une distribution de code transparente. Il ne nécessite pas le recours à des schémas de synchronisation. De plus, un seul langage est à utiliser pour programmer l'entièreté du système.

Il offre en outre la possibilité de vérifier les systèmes créés. Cet aspect est très important parce que les systèmes modélisés sont souvent critiques, c'est-à-dire des systèmes dont les pannes peuvent avoir des effets catastrophiques. Leurs défaillances sont susceptibles d'entraîner directement ou indirectement la perte de vies humaines ou, du moins, des risques majeurs pour le public ou des coûts importants. Il s'agit par exemple de systèmes de contrôle, de sécurité, de surveillance, etc. Ils peuvent dès lors être soumis à des règlements de certification. A titre d'exemple, signalons qu'il existe pour les systèmes de pilotage automatique des avions de transport, un règlement de certification américain FAR25 et son équivalent européen JAR25.

Notons, à titre d'information, qu'un nouveau type de CPs est en cours de développement pour dSL : le PIP-5000. Il s'agit d'un automate fonctionnant sous µC-linux avec un processeur Motorola ColdFire avec 8Mb de RAM.

8. Inconvénients de dSL

L'inconvénient majeur de dSL est qu'il n'est pas, à proprement parlé, modulaire à cause de la contrainte atomique. Une application distribuable peut ne plus l'être par le simple changement de site d'une variable. Par exemple, prenons une méthode définie sur une classe. Si toutes les variables de la classe utilisées dans la méthode sont distribuées sur le même site, la contrainte atomique est respectée. Un grand nombre d'instances de la classe peut donc être défini tant que leurs variables sont distribuées sur le même site. Si maintenant on définit une instance de la classe où une des variables apparaissant dans le code de la méthode est distribuée sur un site différent des autres variables de l'instance, l'application n'est plus distribuable et le code doit être changé. Cet inconvénient a aussi des impacts sur la compilation. En effet, il n'est pas possible de faire une compilation séparée de librairies. Et cela parce que le code généré est dépendant de la localisation et que la localisation est dépendante des objets instanciés dans le programme. Imaginons une méthode définie sur une classe dans une librairie. Si une instance de la classe est ajoutée dans le programme et que cette instance est distribuée de manière différente des autres instances déjà définies, la librairie doit être recompilée. Ceci n'est pas le cas pour des langages tels que C++ ou Java.

9. La syntaxe de dSL

Le concept principal de dSL est la notion d'événement. En réaction à un événement issu de l'environnement, un traitement est exécuté. Un événement est spécifié par le changement de valeur d'une condition booléenne portant sur une ou plusieurs variables globales. Son traitement est défini par une suite d'instructions de base, de lancement de méthode ou de séquence. Par exemple, « `WHEN x < 0 then run_motor()` » déclenchera l'exécution de la méthode `run_motor()` chaque fois que la valeur de `x` passe d'une valeur positive, nulle ou inconnue (cf. *8.5. La valeur UNKNOWN*) à une valeur négative. Le code des méthodes et des séquences est défini ailleurs dans le programme.

La syntaxe complète du langage dSL est reprise à l'*Annexe A*.

9.1. Les identificateurs

En dSL, il est possible de définir des variables, des méthodes, des séquences et des événements. Chacune de ces entités doit avoir un nom, son identificateur, pour être univoquement désignée à l'intérieur du programme. Les identificateurs sont régis par les règles usuelles, à savoir qu'ils peuvent être composés de une ou plusieurs lettres (sans accentuation particulière) ou chiffres mais que le premier symbole doit être une lettre ou le caractère « _ ». Tous les caractères d'un identificateur sont significatifs.

id ├ (lettre | « _ ») (lettre | chiffre)*

9.2. La structure du programme

Un programme dSL est constitué de cinq éléments :

1. Les déclarations des variables globales (et notamment des entrées et sorties),
2. Les définitions des méthodes,
3. Les définitions des séquences,
4. Les définitions des événements,
5. Une initialisation éventuelle.

9.3. Les types de base des variables

Il existe trois types de bases en dSL:

- Les entiers, correspondant au mot syntaxique **INT**. La valeur maximale d'un entier ne peut pas excéder 2^{16}-1.
- Les booléens, correspondant au mot syntaxique **BOOL** de valeur FALSE (0) ou TRUE (1).
- Les entiers de type long, correspondant au mot syntaxique **LONG**. La valeur maximale d'un entier de type long est 2^{32}-1. Ils peuvent être par exemple utilisés pour les temps.

Aucun mécanisme de conversion automatique n'est disponible en dSL. En d'autres termes, pour effectuer par exemple une assignation avec deux variables de type différents, il est nécessaire d'utiliser des primitives de conversion du style « `x := LONG_TO_INT(y)` » qui convertit l'entier de type long y en entier. Ces primitives sont reprises dans la figure 2.5.

Primitives	***Type du paramètre***	***Type du résultat***
LONG_TO_INT	long	entier
LONG_TO_BOOL	long	booléen
BOOL_TO_LONG	booléen	long
BOOL_TO_INT	booléen	entier
INT_TO_LONG	entier	long
INT_TO_BOOL	entier	booléen

Figure 2.5 : Ensemble des primitives de conversion.

9.4. Les types étendus

Il est possible de définir des vecteurs au moyen de la syntaxe suivante :

ARRAY [*nombre_1* **:** *nombre_2* **] OF** *type*

où *nombre_1* et *nombre_2* correspondent aux bornes supérieures et inférieures des index du vecteur et *type*, le type des éléments du vecteur.

Il est aussi possible de définir des classes grâce à la syntaxe suivante :

« **CLASS** *id liste_déclarations* **END_CLASS** »

où *id* est l'identificateur de la classe et *liste_déclarations* l'ensemble des déclarations des variables de la classe (cf. *9.6. Les variables*).

9.5. La valeur UNKNOWN

Toute variable peut prendre la valeur UNKNOWN, c'est-à-dire inconnue, dans le cas par exemple d'une mauvaise opération ou d'un problème de réseau. L'évaluation de n'importe quelle expression dont un des membres est à la valeur UNKNOWN donne la valeur UNKNOWN.

Il est possible de tester si une variable ou une expression est de valeur inconnue grâce à la fonction IS_UNKNOWN qui doit être utilisée de la manière suivante :

« **IS_UNKNOWN** *expression* »

9.6. Les variables

Une variable est un espace mémoire nommé qui peut être utilisé pour contenir une valeur qui peut être modifiée pendant l'exécution du programme. Toute variable doit être déclarée avant d'être utilisée. La forme générale d'une déclaration d'une variable est

« *id* **:** *type* **;** »

où *id* est l'identificateur de la variable et *type* son type.

Remarquons que différentes variables d'un même type peuvent être définies ensemble grâce à la syntaxe suivante où *id1* et *id2* sont les deux variables de même type :

« *id1***,** *id2* **:** *type***;** »

Une variable peut être globale au programme (elle pourra être utilisée n'importe où dans le programme) ou locale à un certain événement, méthode ou séquence (elle ne pourra être utilisée qu'à l'intérieur du code de l'événement, de la méthode ou de la séquence). L'endroit de la déclaration d'une variable définit sa visibilité. Les variables globales sont définies au début du programme (cf. *9.2. La structure du programme*). Les déclarations de celles-ci sont définies grâce à la syntaxe

« **GLOBAL_VAR** *liste_déclarations* **END_VAR »**.

où *liste_déclarations* est une suite de déclaration de variables.

Les variables locales sont définies au début du code d'un événement, d'une méthode ou d'une séquence au moyen de la syntaxe suivante :

« **LOCAL_VAR** *liste_déclarations* **END_VAR »**.

Il n'est pas possible de définir des variables ailleurs qu'en début de programme, d'événement, de méthode ou de séquence.

9.7. Les sites et les entrèes/sorties

Un programme dSL contient la définition des sites sur lesquels il sera distribué. Leur nombre ainsi que la distribution des variables globales sur les différents sites composant le système sont définis par le programmeur dans le programme dSL.

La définition d'un site est effectuée grâce à

« **SITE** *nom_site* **:** *id_site* *liste_variables_globales* **END_SITE** »

où *id_site* est l'identificateur du site et *liste_variales_globales* est la liste des variables assignées à ce site. Pour chaque variable globale du site, il faut définir sa sorte et ses paramètres. Les variables globales peuvent être de trois sortes : interne au programme, liée à une entrée (« **INPUT** ») ou liée à une sortie (« **OUTPUT** »). Les paramètres d'une variable globale d'entrée ou de sortie consistent en trois nombres entiers (*rack* « . » *card* « . » *slot*) permettant de spécifier le type particulier de senseur d'entrée ou de sortie auquel correspond la variable et l'emplacement du dispositif. Le nombre et les intitulés des paramètres sont issus des possibilités offertes par les CP pour les dispositifs d'entrée/sortie (cf. *1. Introduction au langage SL*) et ont été conservés tels quels. La déclaration d'une variable globale d'entrée ou de sortie identifiée par *id_variable* est donc définie comme suit :

« *sorte id_variable* : NOMBRE **.** NOMBRE **.** NOMBRE **;** »

Les variables internes d'un programme ne nécessitent pas de spécifier leur site de distribution. Celle-ci est déterminée par le distributeur en fonction de la localisation des instructions.

Il va de soit que les variables d'entrée ne pourront en aucun cas être assignées. En d'autres termes, elles ne seront jamais le membre de gauche d'une assignation.

9.8. Les opérateurs et les séparateurs

Il existe différents types d'opérateurs en dSL : les opérateurs d'assignation, arithmétiques, relationnels et logiques. A ceux-là s'ajoutent d'autres opérateurs plus spécifiques tels que l'opérateur tilde par exemple. Il y a deux séparateurs en dSL qui sont les séparateurs d'instructions et de commentaires.

a) Opérateur d'assignation

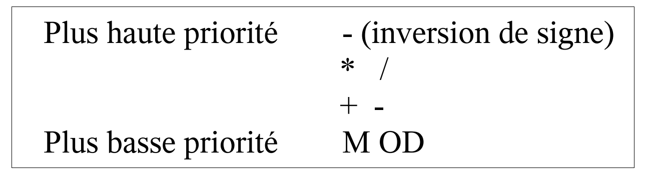
L'opérateur d'assignation « := » est un opérateur binaire qui remplace la valeur de la variable identifiée par son membre de gauche par la valeur de l'expression de son membre de droite (cf. *8.10.a Les instructions d'assignation*).

b) Opérateurs arithmétiques

La figure 2.6 reprend les différents opérateurs arithmétiques de dSL. Ces opérateurs se comportent de la même manière que dans les langages de programmation classiques. Les priorités de ces opérateurs sont celles représentées dans la figure 2.7.

Opérateurs	***Actions***
-	Soustraction et inversion de signe
+	Addition
*	Multiplication
/	Division
MOD	Modulo

Figure 2.6 : Opérateurs arithmétiques dSL.



Plus haute priorité	- (inversion de signe)
	* /
	+ -
Plus basse priorité	M OD

Figure 2.7 : Priorités des opérateurs arithmétiques dSL.

c) Opérateurs relationnels

Les opérateurs relationnels spécifient les relations que les variables peuvent avoir entre elles. Ils sont repris à la figure 2.8 et sont similaires aux opérateurs relationnels traditionnels. La forme générale de l'usage d'un opérateur relationnel est « *expression opérateur_relationnel expression* ».

Opérateurs	***Actions***
>	Plus grand que
>=	Plus grand ou égal à
<	Plus petit que
<=	Plus petit ou égal à
==	Égal à
<>	Pas égal à

Figure 2.8 : Opérateurs relationnels dSL.

d) Opérateurs logiques

Les opérateurs logiques spécifient la manière dont les relations entre les variables peuvent être associées. Les deux opérateurs principaux à disposition en dSL sont le **AND** (« et » logique) et le **OR** (« ou » logique). dSL fournit aussi l'opérateur **NOT** qui inverse la valeur d'une expression logique. Il n'y a pas de OR exclusif (traditionnellement XOR) mais celui-ci peut être évalué par une combinaison des trois opérateurs relationnels existants selon la forme suivante :

```
« ( operateur_1 OR operateur_2 ) AND NOT ( operateur_1 AND operateur_2 ) »
```

Les priorités des opérateurs logiques et relationnels sont reprises dans la figure 2.9 .

Plus haute priorité	**NOT**
	> >= < <=
	==
	AND
Plus basse priorité	**OR**

Figure 2.9 : Priorités des opérateurs logiques et relationnels dSL.

e) Opérateur « . »

L'opérateur point est utilisé pour référencer un élément individuel d'une structure de classe. Il s'utilise à l'intérieur d'une structure, c'est-à-dire dans une méthode de la classe ou un WHEN IN, pour référencer un objet interne à la classe, à savoir une de ses variables. L'usage de « `self` » permet de référencer l'instance courante de la classe (cf. *8.11 Les méthodes* et *8.13 Les événements*).

f) Opérateurs de lancement « LAUNCH » et « <- »

Ces opérateurs permettent de lancer l'exécution asynchrone d'une séquence ou d'une méthode d'une classe sur une instance de cette classe dans une instruction de lancement (cf. *8.10.e Les instructions de lancement*). L'opérateur « <- » permet de spécifier sur quelle instance de la classe la méthode est exécutée.

g) Opérateur « ~ »

L'opérateur tilde permet de référencer une variable dite tildée, c'est-à-dire la copie locale d'une variable (cf. *Code atomique et séquentiel*) . Il s'agit d'un opérateur unaire. La forme générale est

« **~** *id_variable* »

Aucune assignation ne peut être effectuée sur une variable tildée. Aucune référence à une variable tildée ne peut être faite dans le corps d'un WHEN, seulement dans sa condition.

h) Séparateur « ; »

Ce séparateur permet de définir la fin d'une instruction. Il est nécessaire pour tous les types d'instructions.

i) Séparateurs « (* » et « *) »

Ces séparateurs permettent d'introduire des commentaires dans un programme, c'est-à-dire une suite de caractères qui ne sera pas prise en compte lors de la compilation. Ils sont utiles lorsque le programmeur veut commenter son code et n'ont aucun effet sur l'exécution du programme car ils sont ignorés par le compilateur. Les commentaires peuvent être placés partout sauf au milieu d'un mot-clé ou d'un identificateur. Ils ne peuvent être imbriqués.

9.9. Les expressions

Une expression valide dSL est une combinaison correcte d'opérateurs (hormis les opérateurs d'assignation et de lancement), de constantes et de variables. La notion de combinaison correcte est définie par la syntaxe complète de dSL reprise en *Annexe A*. L'évaluation d'une expression se fera suivant les différentes priorités des éléments la composant.

Une expression peut être logique ou arithmétique suivant le type de ses variables et de ses opérateurs. Une expression logique est une combinaison correcte d'opérateurs, de constantes et de variables logiques telle que par exemple « `booleen OR FALSE` ». Une expression arithmétique est une combinaison d'opérateurs, de constantes et de variables arithmétiques telle que « `5+2*3` ».

L'ordre d'évaluation des éléments d'une expression n'est pas spécifié. Le programmeur veillera donc à ne pas se baser sur l'ordre dans lequel les sous-expressions sont évaluées. En l'occurence, l'usage des parenthèses permet de définir de manière précise l'ordre d'évaluation des sous-expressions.

9.10. Les instructions

Selon la définition courante, une instruction est la plus petite partie d'un programme qui peut être exécutée. En dSL, il est possible de différencier différents types d'instructions :
- d'assignation,
- conditionnelle,
- d'itération,
- d'attente,
- et de lancement.

Les instructions conditionnelles correspondent aux instructions IF. Les instructions d'itération sont déterminées par WHILE. Les instructions d'attente sont les instructions WAIT. Celles de lancement, les instructions LAUNCH. En dSL, toute instruction se termine par un « **;** ».

a) Les instructions d'assignation

La forme générale d'une assignation est

« *nom_variable* **:=** *expression* **;** »

où le membre de droite peut être une simple constante ou une expression plus complexe. La cible, c'est-à-dire la partie gauche de l'assignation, doit être l'identificateur d'une variable mais pas d'un événement, d'une méthode, d'une séquence ou une constante.

dSL ne permet pas les assignations multiples. Une instruction d'assignation ne porte que sur une variable qui doit être définie dans le scope de l'instruction. Cette variable doit être de type interne ou de sortie (cf. *9.7. Les sites*).

b) Les instructions conditionnelles

La forme générale d'une instruction conditionnelle est

« **IF** *condition* **THEN** *liste_instructions_1* **ELSE** *liste_instructions_2* **END_IF ;** »

où *condition* est une expression logique et *liste_instructions_1* et *liste_instructions_2* sont des blocs d'instructions qui sont exécutés si la condition est respectivement vraie ou fausse lors de l'exécution de l'instruction. Il est possible de ne pas spécifier de traitement dans le cas où la condition est fausse grâce à la forme générale suivante

« **IF** *expression* **THEN** *liste_instructions* **END_IF ;** »

c) Les instructions d'itération

La forme générale d'une instruction d'itération est

« **WHILE** *condition* **DO** *liste_instructions* **END_WHILE ;** »

La construction WHILE permet d'exécuter une suite d'instructions tant qu'une condition est vraie. Lorsque le contrôle arrive sur une instruction WHILE, la condition est évaluée. Si celle-ci est fausse, l'instruction suivant le « **END_WHILE »** est exécutée. Si celle-ci est vraie, la liste d'instructions est exécutée et la condition est réévaluée.

d) Les instructions d'attente

L'instruction WAIT permet de spécifier un arrêt dans l'exécution d'une séquence. Cet arrêt est défini soit sur un intervalle de temps, soit sur une condition.

Dans le premier cas, la forme générale est

« **WAIT** *constante* **;** »

où *constante* est le temps en milisecondes pendant lequel l'exécution de la séquence sera retardée.

Dans le second cas, la forme générale est

« **WAIT** *condition* **;** »

où *condition* est une expression logique. L'exécution de la séquence est suspendue juqu'à ce que la condition soit vraie.

e) Les instructions de lancement

Les instructions de lancement sont définies au moyen des opérateurs« **<-** » et « **LAUNCH** » . Les formes générales d'une instruction de lancement d'une méthode sont

« *instance_classe* **<-** *id_méthode***(***paramètres***)** **;**»

« **LAUNCH** *instance_classe* **<-** *id_méthode***(***paramètres***)** **;** »

où *instance_classe* est l'identificateur de l'instance de la classe sur laquelle va s'exécuter la méthode, *méthode* est l'identificateur de la méthode et *paramètres*, les paramètres transmis à la méthode.

Dans le premier cas, c'est-à-dire sans LAUNCH, l'exécution de la méthode est synchrone et la fin de celle-ci est attendue avant de continuer l'exécution des instructions suivantes.

Dans le second cas, l'exécution de la méthode se fait de manière asynchrone. L'exécution des instructions suivant l'appel à la méthode se fait sans attendre la fin de la méthode.

Le lancement (toujours asynchrone) d'une séquence se fait par l'intermédiaire de LAUNCH selon la syntaxe suivante :

« LAUNCH *id_séquence***(***paramètres***)** **;»**

9.11. Les méthodes

La forme générale de la définition d'une méthode est

« **METHOD** *classe* **::** *id_méthode* **(***paramètres***)** *liste_instructions* **END_METHOD** »

où *classe* est l'identificateur de la classe à laquelle se rapporte la méthode, *id_méthode* est l'identificateur de la méthode, *paramètres* sont les éventuels paramètres de la méthode, et *liste_instructions* est le code de la méthode contenant éventuellement une définition de variables locales en début de bloc. Les paramètres d'une méthode consistent en une suite de déclarations de variables séparées par des virgules. Lors d'un appel de méthode, l'ordre et le type des paramètres doit respecter ceux définis par la déclaration. Cette dernière doit être unique et antérieure à tout appel.

Dans le code de la méthode, seules les variables locales à la méthode (y compris les paramètres) et les variables globales peuvent être utilisées. Il est possible de référencer les variables de l'instance courante de la classe via « `self` » (cf. *8.8.e L'opérateur « . »*). De plus, le code d'une méthode doit respecter la contrainte atomique (cf. *6. Code atomique et séquentiel*).

Signalons aussi qu'en dSL, les appels récursifs ne sont pas permis pour des raisons de vérification du code à cause du nombre d'appels qui pourrait être infini.

9.12. Les séquences

La forme générale d'une séquence est

« **SEQUENCE** *id_séquence* **(***paramètres***)** *liste_instructions* **END_SEQUENCE** »

où *id_sequence* est l'identificateur de la séquence, autrement dit son nom, paramètres, l'éventuelle liste des paramètres de la séquence et *liste_instructions*, une suite d'instructions précédée éventuellement d'un bloc de déclaration des variables locales de la séquence (cf. *9.6. Les variables*). Dans le code de la séquence, seules les variables locales à la séquence (y compris ses paramètres) et les variables globales peuvent être utilisées.

9.13. Les événements

La forme générale d'une instruction d'événement est

« **WHEN** *condition* **THEN** *liste_instructions* **END_WHEN** »

où *condition* est l'expression logique définissant l'événement et *liste_instructions* est la liste des instructions constituant le traitement de l'événement. Elles sont exécutées lorsque la condition de l'événement passe de faux à vrai. Celles-ci sont éventuellement précédées d'un bloc de déclaration de variables locales. Le code d'un événement doit respecter la contrainte atomique (cf. *6. Code atomique et séquentiel*).

La syntaxe suivante

« **WHEN IN** *id_type condition* **THEN** *liste_instructions* **END_WHEN** »

permet de définir un événement sur un type particulier de variable. En d'autres termes, l'événement sera défini pour toutes les variables de type *id_type*. Il est possible de référencer les variables de l'instance courante de la classe via « `self` » (cf. *8.8.e L'opérateur « . »*).

9.14. L'initialisation

En dSL, une initialisation est possible. Pour cela, il suffit de déclarer une séquence sans paramètre identifiée par « **MAIN( )** ». Cette séquence sera exécutée dès l'initialisation du système.

10. Exemples

10.1. Les identificateurs

Les identificateurs permis sont les chaînes de caractères respectant la règle syntaxique présentée plus haut (cf. *9.1.Les identificateurs*). Ils peuvent contenir des lettres, des chiffres ou le caractère « _ » sans restriction de longueur. Le premier caractère ne peut pas être un chiffre. Les majuscules sont permises mais pas les accentuations. La figure 2.9 reprend un ensemble d'exemples d'identificateurs corrects et incorrects contenant uniquement les caractères « a », « b », « c », « d », « A », « C », « D », « _ », « 1 », « 2 », « 3 » et « 4 ».

Identificateurs corrects		***Identificateurs incorrects***
`abcd`	`AbcD`	`1234`
`_abcd`	`_AbCd`	`1abcd`
`a1b2c3d4`	`_1234`	`àbcd`
`_ab12cd34`		`abc_d`

Figure 2.9 : Exemples d'identificateurs corrects dSL.

10.2. La structure d'un programme

Un programme dSL doit respecter un certain nombre de contraintes quant à l'ordre de déclaration de ses éléments : classes, variables, sites, méthodes, séquences, événements (cf. *9.2. La structure du programme*). La figure 2.10 reprend le squelette général d'un programme dSL correct. Pour de plus amples précisions, le programme dSL de l'étude de cas peut être consulté en *Annexe K*. Remarquons que l'ordre des événements, méthodes et séquences peut être différent.

```
CLASS (*déclaration des classes éventuelles*)
...
END_CLASS (*fin de la dernière déclaration de classe*)

GLOBAL_VAR (*déclaration des variables globales*)
...
END_VAR (*fin de la déclaration des variables globales*)

SITE (*déclaration des sites*)
...
END_SITE (*fin de la dernière déclaration de site*)

METHOD (*déclaration des méthodes éventuelles*)
...
END_METHOD (*fin de la dernière déclaration de méthode*)

SEQUENCE (*déclaration des séquences*)
...
END_SEQUENCE (*fin de la dernière déclaration de séquence*)

WHEN (*déclaration des événements (WHEN) éventuels*)
...
END_WHEN (*fin de la dernière déclaration d'événement*)

WHEN IN (*déclaration des événements (WHEN IN) éventuels*)
...
END_WHEN (*fin de la dernière déclaration d'événement*)
```

Figure 2.10 : Squelette d'un programme dSL.

10.3. Les variables globales

La figure 2.11 présente un extrait de programme dSL contenant la déclaration de quelques variables globales, chacune de différents types. Les deux premières sont de type entier, la troisième de type booléen, la quatrième est un entier long et les dernières sont des vecteurs d'entiers et de booléens. La déclaration des variables globales est placée en début de programme (cf. *9.2. La structure du programme*).

```
GLOBAL_VAR
    entier1, entier2 : INT;
    booleen : BOOL;
    long : LONG;
    vecteur_entier : ARRAY [0:10] of INT;
    vecteur_booleen : ARRAY [2:5] of BOOL;
END_VAR
```

Figure 2.11 : Exemple de déclaration de variables globales.

Le dernier type de variables qui peut être utilisé est la classe. La figure 2.12 présente la déclaration d'une classe `voiture` contenant deux variables, l'une entière et l'autre booléenne correspondant respectivement à la vitesse et l'état du véhicule. La déclaration des classes doit se trouver avant celle des variables globales (cf. *9.2. La structure du programme*). Dans la figure 2.12, deux instances de la classe sont définies globalement.

```
CLASS voiture
      vitesse : INT;
      marche : BOOL;
END_CLASS

GLOBAL_VAR
      v1, v2 : voiture ;
END_VAR
```
Figure 2.12 : Exemple de déclaration de classe.

De manière similaire, il est possible de définir des variables locales à une méthode, une séquence ou un événement. La déclaration de variables locales est illustrée aux exemples concernant ces différentes constructions.

10.4. Les sites

Les variables globales peuvent être réparties sur les différents sites du système. La figure 2.13 reprend un exemple de répartition de variables globales. Pour chacune d'elles qui correspondent à un dispositif d'entrée ou de sortie, la sorte et les paramètres doivent être spécifiés.

```
GLOBAL_VAR
  entree1, entree2 : INT;
  sortie1, sortie2 : BOOL;
  interne1 : LONG;
END_VAR

SITE site1 : 1
   INPUT entree1 : 0.0.1 ;
   OUTPUT sortie1 : 1.1.1 ;
END_SITE

SITE site2 : 2
  INPUT entree2 : 0.0.1 ;
  OUTPUT sortie2 ; 1.0.1 ;
END_SITE
```
Figure 2.13 : Exemple de définition de sites.

Dans le cas d'une classe, il est nécessaire de définir le type de toutes les variables d'une instance. Un exemple de répartition d'une instance de classe est repris à la figure 2.14.

```
CLASS AB
     a,b : INT;
END_CLASS

GLOBAL_VAR
     _ab : AB;
END_VAR

SITE site1 : 1
     INTERNAL _ab.a ;
END_SITE

SITE site2 : 2
     OUTPUT _ab.b : 1.0.1 ;
END_SITE
```

Figure 2.14 : Exemple de définition de site avec une classe.

10.5. Les instructions d' assignations

La figure 2.18 reprend un ensemble de différentes instructions d'assignation d'une variable correctes. Rappelons que la variable de destination doit être une variable interne, de sortie ou locale et qu'il n'y a pas de conversion automatique en dSL. Les types de chacune des opérandes de l'expression de droite doivent correspondre avec, si besoin est, le recours aux primitives de conversion (cf. *9.3. Les types de base des variables*). En ce qui concerne les vecteurs, l'indice spécifié entre « **[** » et « **]** » doit être compris entre les bornes inférieure et supérieure de sa déclaration.

```
entier := 0 ;
entier := 1 + 2 - 3 MOD 10;
entier := entier + 1 ;
entier := LONG_TO_INT(long) + BOOL_TO_INT(booleen) ;

booleen := TRUE ;
booleen := LONG_TO_BOOL(long) OR INT_TO_BOOL(entier);

vecteur[1] := 10;
vecteur[3] := vecteur[2] + vecteur[1];
```

Figure 2.18 : Exemples d'assignation dSL.

Lorsqu'une variable d'une classe est assignée, il est nécessaire de préciser l'instance de la classe sauf en ce qui concerne les instructions de code des méthodes de la classe (cf. *9.11. Les méthodes*). La figure 2.19 reprend un exemple de déclaration d'une classe et d'assignations de ses valeurs.

```
CLASS AB
     a,b : INT;
     c : BOOL;
END_CLASS

GLOBAL_VAR
     _ab : AB;
END_VAR

SITE site1 : 1
     INTERNAL _ab.a ;
     OUTPUT _ab.b : 1.0.1 ;
END_SITE

...

_ab.a := 10;
_ab.b := _ab.a + _ab.b;
_ab.c := INT_TO_BOOL(_ab.b);
```

Figure 2.19 : Exemple d'assignations sur une classe.

10.6. Les instructions conditionnelles

La figure 2.20 montre différents exemples d'instructions conditionnelles. La première définit que si `entier` est supérieur à 10, `booleen` est mis à TRUE. La deuxième ajoute le fait que si `entier` est inférieure ou égal à 10, alors `booleen` est mis à FALSE. La troisième montre un exemple d'instructions conditionnelles imbriquées. Si la condition est de valeur inconnue, l'instruction conditionelle n'a pas d'effet.

```
GLOBAL_VAR
      entier : INT;
      booleen : BOOL;
END_VAR

...

IF entier>10 THEN
     booleen := TRUE;
     ...
END_IF;

IF entier>10 THEN  booleen := TRUE;
ELSE  booleen := FALSE;
END_IF;

IF entier>10 THEN
   IF booleen==FALSE THEN booleen:=TRUE; END_IF;
ELSE
   IF booleen==TRUE THEN booleen:=FALSE; END_IF;
END_IF;
```

Figure 2.20 : Exemple d'instructions conditionnelles dSL.

10.7. Les instructions d'itération

Une instruction d'itération est définie sur une expression booléenne. Dans la figure 2.21, le WHILE porte sur `booleen`. L'exécution de ce WHILE aura pour effet d'augmenter la valeur d'`entier` jusqu'à ce que celui-ci soit plus grand que 10. Remarquons qu'une définition équivalente mais plus courte du WHILE est « `WHILE entier<=10 DO entier := entier + 1 ; END_WHILE ;` ». Bien entendu, les instructions d'itérations peuvent être imbriquées de la même manière que les instructions conditionnelles. Si la condition a la valeur inconnue, l'instruction d'itération n'a aucun effet.

```
GLOBAL_VAR
      entier : INT;
      booleen : BOOL;
END_VAR

...

IF entier>10 THEN booleen := TRUE;
ELSE booleen := FALSE END_IF;

WHILE booleen DO
      entier := entier+1;
      IF entier>10 THEN
           booleen := TRUE;
      END_IF
END_WHILE;
```

Figure 2.21 : Exemple d'instructions d'itération dSL.

10.8. Les instructions d'attente

Les instructions d'attente peuvent porter soit sur un intervalle de temps entier défini par une expression arithmétique entière, soit sur une condition définie par une expression booléenne. La figure 2.22 présente différents exemples d'instructions d'attente correctes.

```
WAIT 10 ;
WAIT entier1 ;
WAIT ( entier1 + entier2 ) ;

WAIT booleen ;
WAIT ( entier1 == entier2 ) ;
```

Figure 2.22 : Exemple d'instructions d'attente.

10.9. Les méthodes et les instructions de lancement de méthode

Les méthodes sont définies par rapport à une classe. Elles s'exécutent toujours sur une instance de la classe. Les variables de l'instance sont accédées au moyen du mot « `self` ». La figure 2.23 présente deux exemples de méthode d'une classe `voiture`.

```
CLASS voiture
      vitesse : INT;
      marche : BOOL;
END_CLASS

GLOBAL_VAR
      v1 : voiture ;
END_VAR

METHOD voiture :: accelerer(ajout : INT)
     self.vitesse := self.vitesse + ajout;
END_METHOD

METHOD voiture :: freiner()
     self.vitesse := 0 ;
END_METHOD
```

Figure 2.23 : Exemple de déclaration de classe.

La figure 2.24 présente différents exemples d'appels de ces méthodes (cf. *9.10.e) Les instructions de lancement*). Le premier met `v1.vitesse` à 0. Le deuxième lui ajoute 2. Les deux derniers montrent comment il est possible d'utiliser des variables dans les paramètres d'un appel à une méthode. Le troisième exemple présente un appel de fonction synchrone. `plus` sera mis à 0 une fois que la méthode sera terminée. Par contre, le quatrième exemple présente un appel asynchrone. L'exécution va se poursuivre sans attendre la fin de la méthode.

```
LAUNCH v1<-freiner();


LAUNCH v1<-accelerer(2);


plus : INT;
...
plus := 2;
v1<-accelerer(plus);
plus := 0;


...
plus := 2;
LAUNCH v1<-accelerer(plus);
plus := 0;
```

Figure 2.24 : Exemples d'appels de méthodes.

Il est possible de définir un ensemble de variables locales à une méthode. La figure 2.25 présente un exemple de méthodes à deux paramètres qui définit localement une variable temporaire. Remarquons qu'il n'y a pas de retour de valeur.

```
METHOD bidon :: switch( a : INT, b : INT)
       LOCAL_VAR
             temp : INT ;
       END_VAR
       ...
       temp := a ;
       a := b ;
       b := temp ;
       ...
END_METHOD
```

Figure 2.25 : Exemple de méthode à deux paramètres avec variables locales.

Il est important de ne pas oublier que le code d'une méthode est atomique. Il doit donc être distribuable sur un seul site. La figure 2.26 montre un extrait de programme contenant différents exemples de méthodes correctes. La figure 2.27 présente un exemple de méthode qui ne respecte pas la contrainte atomique et son équivalent qui la respecte.

```
CLASS ab
   a,b := INT;
END_CLASS

CLASS cd
   c,d : INT;
END_CLASS

GLOBAL_VAR
   _A, _B : ab ; (*_A et _B sont des instances de la classe ab)
   _C : cd ;
   e,f,g : INT;
END_VAR

SITE site1
   INPUT _A.a : 0.0.1 ;
   OUTPUT _A.b : 1.0.1 ;
   INPUT _C.c : 0.0.1 ;
   INPUT _C.d : 1.1.1 ;
END_SITE

SITE site2
   INPUT _B.a : 0.0.1 ;
   OUTPUT _B.b : 1.0.1 ;
END_SITE

METHOD ab::assignA(entier : INT)
   self.a := entier;
END_METHOD

METHOD ab::switch()
   LOCAL_VAR
      temp : INT;
   END_VAR
   temp := self.a;
   self.a := self.b;
   self.b := temp;
END_METHOD

METHOD cd::assignCD(entierC : INT, entierD : INT)
   self.c := entierC;
   self.d := entierD;
END_METHOD

METHOD cd::assignCtoF()
   self.c := f;
END_METHOD
```

Figure 2.26 : Programme dSL avec exemples de méthodes.

Méthode incorrecte	***Méthode correcte***

`METHOD cd::assignCtoG()`
`   self.c := g; (* c:site1, g:site2 *)`
`END_METHOD`
	`METHOD cd::assignCtoG()`
	`   self.c := ~g;`
	`END_METHOD`

Figure 2.27 : Exemple de méthode incorrecte et de sa correction.

10.10. Les séquences et les instructions de lancement de séquence

Une séquence est une suite d'instructions identifiée par un identificateur et des paramètres et ne respectant pas nécessairement la contrainte atomique (cf. *6. Code atomique et séquentiel*). La figure

2.29 reprend un exemple de séquence correcte. La séquence est constituée de différents blocs qui seront exécutés sur les sites spécifiés en commentaires. L'instruction permettant d'exécuter de manière asynchrone cette séquence est « `LAUNCH distri();` » (cf. *9.10.e) Les instructions de lancement*).

```
GLOBAL_VAR
   a,b,c,d,e,f : INT;
END_VAR

SITE site1
   INPUT a : 0.0.1 ;
   OUTPUT b : 1.1.1 ;
END_SITE

SITE site2
   INTERNAL c ;
   INTERNAL d ;
END_SITE

SITE site3
   INTERNAL e ;
   INTERNAL f ;
END_SITE

SEQUENCE distri()
   LOCAL_VAR
      l : INT;
   END_VAR

   IF a>0 THEN b:=1; ELSE b:=0; END_IF (*site1*)
   l:=c; c:=d; d:=l; (*site2*)
   b:=a+b+l; (*site1*)
   e:=f; (*site3*)
   l:=a; b:=l; (*site1*)
END_SEQUENCE
```

Figure 2.29 : Exemple de séquence dSL.

La séquence de la figure 2.30 prend deux paramètres. Elle est distribuée sur le site 1 et utilise une instruction d'attente.

```
GLOBAL_VAR
   h : INT;
   i : BOOL;
END_VAR

SITE site1
   INPUT a : 0.0.1 ;
   OUTPUT b : 1.1.1 ;
END_SITE

SEQUENCE distri(x:INT,y:BOOL)
   wait(i OR y);
   h:=x;
END_SEQUENCE
```

Figure 2.30 : Exemple de séquence dSL à deux paramètres.

10.11. Les événements

Un événement est défini sur une condition et une suite d'instructions respectant la contrainte atomique. Les différentes méthodes pour relâcher la contrainte atomique ont été détaillées dans *6. Code atomique et séquentiel*. La figure 2.31 présente un ensemble d'événements définis de manière correcte. Les sites de distribution de chacun des événements ont été précisés en commentaires.

```
CLASS CD
   c,d : INT;
END_CLASS

GLOBAL_VAR
   a,b : INT;
   cd : CD;
END_VAR

SITE site1
   INPUT a : 0.0.1 ;
   OUTPUT b : 1.1.1 ;
END_SITE

SITE site2
   INTERNAL cd.c ;
   INTERNAL cd.d ;
END_SITE

METHOD CD::cEGALd()
   self.c := self.d ;
END_METHOD

SEQUENCE bEGALa()
   b := a ;
END_SEQUENCE

WHEN a>0 THEN (*site1*)
   b:=1;
END_WHEN

WHEN a<0 THEN (*site1*)
   LAUNCH bEGALa();
END_WHEN

WHEN c==0 THEN (*site2*)
   cEGALd(); (*appel synchrone sur son propre site*)
END_WHEN

WHEN a==0 THEN (*site1*)
   LAUNCH cEGALd(); (*appel asynchrone sur un site distant*)
END_WHEN

WHEN ~a==0 THEN (*site2*)
   cEGALd();
END_WHEN
```

Figure 2.31 : Exemple de déclarations d'événements.

Il est possible de définir un événement sur un type de variable grâce à WHEN IN. La figure 2.32 reprend un exemple d'utilisation de cette construction.

```
WHEN IN CD self.c==0 THEN
    LAUNCH self<-cEGALd();
END_WHEN
```
Figure 2.32 : Exemple de WHEN IN.

11. La sémantique

Dans ce chapitre, nous allons présenter la sémantique du langage dSL. Après avoir introduit certaines notations utiles, nous nous attacherons à la notion de distribution maximale. Ensuite, nous nous pencherons sur quelques définitions importantes telles que celles de système à transitions étiquetées ou d'environnement d'exécution. Pour finir, nous détaillerons les règles de sémantique opérationnelle structurelle de dSL.

La sémantique qui suit est tirée de [**DMM03(2)**]. En fait, il s'agit de la sémantique d'un sous-ensemble de dSL. En effet, pour des raisons de simplicité, certaines restrictions ont été faites. Les méthodes sont supposées inline. Ce qui implique que les appels récursifs ne sont pas permis. Dès lors, toutes les variables sont considérées comme déclarées globalement. Les séquences et les instructions de lancement ne sont pas considérées. Il est possible d'exprimer une séquence ou un LAUNCH de manière équivalente par un code utilisant des WHENs et des tildes (cf. [**DMM03(2)**]). Pour plus de précisions, se référer à ce document ou à la sémantique complète de dSL à l'*Annexe B*.

11.1. Notations utiles

Tout d'abord, il est nécessaire de préciser les notations qui seront utilisées dans la suite:

- *Var(P),* l'ensemble des variables non tildées apparaissant dans le programme *P;*
- $Var^{in}(P)$, l'ensemble des variables de *Var(P)* qui correspondent à des variables d'entrées;
- $Var^{out}(P)$, l'ensemble des variables de *Var(P)* qui correspondent à des variables de sorties;
- $Var^{T}(P)$, l'ensemble des variables de *Var(P)* qui correspondent à des variables internes;
- *Var(w)*, l'ensemble des variables non tildées apparaissant dans le when *w;*
- *Var(e)*, l'ensemble des variables, tildées ou non, apparaissant dans l'expression *e*;
- $<v(P)$, l'opérateur de l'ordre dans lequel les variables du programme *P* sont déclarées. Cet ordre sera utilisé pour déterminer l'ordre dans lequel les variables d'entrées et de sorties seront respectivement lues ou mises à jour;

- *When(P)*, l'ensemble des when apparaissant dans le programme *P*;
- $<w(P)$, l'opérateur de l'ordre dans lequel les when d'un programme *P* sont déclarés. Cet ordre sera utilisé pour déterminer l'ordre dans lequel les WHEN seront exécutés;

- *Cond(w)*, la condition du WHEN *w*;
- *Body(w)*, la liste d'instructions du WHEN *w*;
- $OldCond(W) = \{oldcond_w \mid w \in W\}$, avec $oldcond_w$, l'évaluation précédente de *Cond(w);*

- $\sim X$, l'ensemble des variables tildées corespondant à X, $\sim X = \{ \sim x \mid x \in X \}$.

De plus, pour définir la sémantique de dSL, les expressions suivantes sont ajoutées pour décrire des traitements internes.

- INPUT *id*, représente la lecture de la variable d'entrée *id.*

- OUTPUT *id*, représente l'écriture de la variable de sortie *id*.
- BCAST *id*, représente l'envoi de la variable *id* à tous les sites d'exécution.
- MSG, représente le traitement des messages stockés dans une file FIFO, c'est-à-dire First In First Out.

11.2. Distribution maximale

dSL permet de modéliser des systèmes distribués. La distribution d'un programme dSL correct est la partition de l'ensemble des variables du programme respectant les contraintes atomiques imposées par les WHEN. Notons que si deux variables apparaissent non tildées dans un WHEN, alors elles doivent être nécessairement distribuées sur le même site. Pour rappel, les méthodes sur lesquelles portent aussi la contrainte atomique, sont supposées inline.

Definition 1 (Distribution d'un programme dSL correct) La distribution d'un programme dSL *P* est un partitionnement $D = \{V_1, V_2, \ldots V_n\}$ de l'ensemble *Var(P)* et est correct si et seulement si

$$\forall\ w \in When(P)\ \forall v_1,v_2 \in Var(w), \exists V_i \in D : \{v_1,v_2\} \subseteq V_i$$

♦

L'ensemble de toutes les distributions correctes d'un programme *P* est notée $\mathcal{D}_P$.

Le comportement d'un programme dSL est défini par la distribution de ses variables et de son code. La contrainte atomique impose que certaines variables et instructions soient distribuées sur un site précis (cf. *6. Code atomique et séquentiel*). En ce qui concerne les autres, elles peuvent être réparties sur un des sites composant le système. Nous allons introduire ici la notion de *distribution maximale* qui représente la configuration la plus libérale possible pour un programme dSL donné. Il a été prouvé dans [**DMM04**] que la *distribution maximale* contient toutes les traces de n'importe quelle distribution *D* de $\mathcal{D}_P$. Dès lors, vérifier la sécurité d'un système sur cette distribution impliquera la sécurité pour tout autre distribution.

Pour définir la notion de *distribution maximale*, il faut pouvoir comparer deux distributions. Nous introduisons donc un ordre partiel sur les distributions, c'est-à-dire une hiérarchie.

Definition 2 (Hiérarchie de distribution) Soit $D = \{V_1, V_2, \ldots V_n\}$ et $D' = \{V'_1, V'_2, \ldots V'_n\}$ deux distributions d'un programme dSL *P*. La distribution *D'* est un raffinement de la distribution *D*, noté $D' \preccurlyeq D$, si

$$\forall V'_i \in D', \exists V_i \in D : V'_i \subseteq V_i$$

♦

Definition 3 (Distribution maximale d'un programme dSL correct) La distribution maximale d'un programme dSL correct *P* est la distribution D_{max} tel que

$$\nexists\ D \neq D_{max} : D_{max} \preccurlyeq D$$

♦

11.3. Définitions préliminaires

Avant de définir la sémantique opérationnelle structurelle de dSL, il faut introduire certaines définitions préliminaires. La sémantique sera définie en termes de systèmes à transitions étiquetées.

Pour rappel, un système à transitions est défini par un ensemble d'états et par une relation de transition entre ces états. Les systèmes à transitions étiquetées sont des systèmes à transitions où celles-ci sont accompagnées d'un symbole appelé étiquette.

Definition 4 (Système à transitions étiquetées) Un système à transitions étiquetées L est un tuple $(Q, q^0, \Sigma, \rightarrow)$ avec

- Q, un ensemble d'états,
- q^0 ($\in Q$), l'état initial,
- Σ, l'ensemble des symboles du langage (appelé alphabet), avec $\tau \notin \Sigma$ (τ est une action interne),
- $\rightarrow \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$, est la relation de transition.

♦

Étant donnés deux états $q, q' \in Q$, pour tout symbole $a \in (\Sigma \cup \{\tau\})$, nous notons $q \rightarrow^a q'$ si $(q,a,q') \in \rightarrow$ et pour tout $w = a_1 \cdot a_2 \cdot ... \cdot a_n \in (\Sigma \cup \{\tau\})^*$, $q \rightarrow^w q'$ s'il existe une séquence de transitions $q_0 \rightarrow^{a_1} q_1 ... \; q_{n-1} \rightarrow^{a_n} q_n$ avec $q = q_0$ et $q' = q_n$.

Étant donné un programme dSL P correct et une distribution $D \in \mathcal{D}_P$, nous pouvons définir l'environnement distribué dans lequel P sera exécuté. En effet, en vertu des contraintes atomiques, le partitionnement des variables du programme P donnée par D impose une répartition de l'ensemble des WHEN de P.

Définition 5 (Contexte d'exécution distribué) Etant donnés un programme dSL P et une distribution $D = \{ V_1, V_2, ... V_n \} \in \mathcal{D}_P$, le contexte d'exécution distribué de P est défini comme suit

$$E^P_D = ((V_1,W_1), (V_2,W_2), ... , (V_n,W_n))$$

où chaque $W_i = \{ w \in When(P) \mid Var(w) \subseteq V_i \}$.

♦

Dans la suite, nous appelerons chaque (V_i,W_i) un contexte d'exécution local et nous le dénoterons par $(E^P_D)_i$. Notons que, puisque D est une distribution de P, nous avons $\cup_{i \in [1...n]} W_i = When(P)$ et $\forall i,j \in [1... n], (i \neq j) \Rightarrow (W_i \cap W_j = \emptyset)$.

Il est aussi nécessaire d'introduire des fonctions auxiliaires suivantes. Les deux premières fonctions qui suivent construisent des listes d'instructions correspondant respectivement à la lecture des entrées et à l'écriture des sorties des variables d'un ensemble V donné.

Définition 6 (Lecture des entrées, Ecriture des sorties) Etant donné un ensemble de variables V, nous définirons $Sample(V, <_V)$, respectivement $Write(V,<_V)$, comme une liste d'instructions réalisant la lecture des variables d'entrée, respectivement l'écriture des variables de sortie, de V dans l'ordre spécifié par $<_V$, comme suit

$$Sample(V, <_V) = \text{INPUT } (v_1); \text{INPUT } (v_2); ... \text{INPUT } (v_k);$$
$$Write(V,<_V) = \text{OUTPUT } (v_1); \text{OUTPUT } (v_1); ... \text{OUTPUT } (v_k);$$

avec $\forall i \in [1...k-1]$, $v_i <_V v_{i+1}$ et $\cup_{i \in [1...n]} v_i = V$.

♦

La définition suivante construit une liste d'instructions correspondant à un traitement des WHENs d'un ensemble *W*.

Définition 7 (Traitement des WHENs) Etant donné un ensemble de WHENs *W*, nous définirons $Treat(W,<_W)$ comme une liste d'instructions réalisant le traitement de tous les WHENs de *W*, selon l'ordre défini par $<_W$, comme suit

$$Treat(W,<_W) = \omega_1, \omega_2, .. \omega_{|W|}$$

avec $\forall i \in [1...|W|-1]$, $\omega_i <_W \omega_{i+1}$ et $\cup_{i\in[1...n]} \omega_i = W$.

♦

11.5. La sémantique opérationnelle structurelle

De manière informelle, le comportement d'un programme dSL peut être vu comme la composition parallèle de *n* processus, un pour chaque site, communiquant avec l'environnement. Chaque processus S_i gère un ensemble de variables V_i et communique avec les autres processus par des canaux FIFO, c'est-à-dire First In First Out, pour signaler entre autre la valeur d'une variable tildée.

En réalité, chaque S_i est la boucle infinie *Input-Process-Ouput* détaillée précédemment, avec *Input* correspond à $Sample(V_i, <_V)$, *Output* à $Write(V_i,<_V)$ et *Process* correspondant au traitement des messages et des WHENs.

Nous commencerons par définir l'état global d'un programme dSL.

Définition 8 (Etat global d'un programme dSL correct) Etant donnée une distribution $D = \{V_1, V_2, ... V_n\}$ d'un programme dSL *P*, l'état global de *P* peut être défini comme suit

$$G \equiv ((\omega_1, \nu_1, \varnothing_1), ... , (\omega_\nu, \nu_n, \varnothing_n))$$

avec $\forall i \in [1..n]$, $(\omega_i, \nu_i, \varnothing_i)$ l'état local du processus *i* où

- ω_i est l'espace de travail, c'est-à-dire les séquences d'instructions restant à exécuter,
- ν_i est une fonction de valuation renvoyant vrai ou faux et définie sur les variables globales du processus *i*, les copies locales des variables tildées et les variables contenant les anciennes valeurs des conditions des WHENs,
- $\varnothing_i$ est le canal FIFO de communication.

♦

L'ensemble des états globaux d'un programme dSL *P*, étant donnée une distribution *D* est noté $\mathcal{G}_D^P$.

Maintenant, il est possible de décrire les règles de sémantique qui définiront la relation de transition des systèmes à transitions labellisées.

La première règle correspond à la sémantique d'interleaving. Par exemple, si, dans un état local, une transition peut être faite, alors à partir de n'importe quel état global contenant cet état local, la même transition peut être prise en ne modifiant que l'état local.

[Interleaving]

$$\frac{(E_D^P)_i \vdash (\omega_i, \nu_i, \phi_i) \xrightarrow{a} (\omega'_i, \nu'_i, \phi'_i)}{E_D^P \vdash ((\omega_1, \nu_1, \phi_1), ..., (\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n)) \xrightarrow{a} ((\omega_1, \nu_1, \phi_1), ..., (\omega'_i, \nu'_i, \phi'_i), ..., (\omega_n, \nu_n, \phi_n))}$$

$$\forall a \in \{\tau\} \cup \Sigma$$

La deuxième règle de sémantique est celle du broadcast. Lorsqu'une variable tildée change de valeur, le site doit transférer ce changement de valeur à tous les autres sites et spécialement aux sites possédant une copie locale de cette variable. Il s'agit d'une τ-transition. Tous les états locaux des sites concernés seront mis à jour avec le nouveau message.

[Broadcast]

$$E_D^P \vdash ((\omega_1, \nu_1, \phi_1), ..., (\texttt{BCAST(x)};\omega_i, \nu_i, \phi_i), ..., (\omega_n, \nu_n, \phi_n)) \xrightarrow{\tau} ((\omega_1, \nu_1, \phi'_1), ..., (\omega_i, \nu_i, \phi'_i), ..., (\omega_n, \nu_n, \phi'_n))$$

$$\text{où } \forall j \in [1..n] : \phi'_j = \phi_j \cdot (x, \nu_i(x), i)$$

A l'inverse des deux premières règles qui étaient des règles globales, les règles suivantes sont des règles locales. La première de ces règles correspond au commencement d'un nouveau cycle. Lorsqu'un espace de travail est vide, un nouveau cycle I*nput-Process-Output* est exécuté. Cette règle définit le comportement cyclique de chaque processus.

[Début Cycle]

$$(E_D^P)_i \vdash (\varepsilon, \nu_i, \phi_i) \xrightarrow{\tau} (Sample(Var^{in}(P) \cap V_i, <_V); Treat(W_i, <_W); \texttt{MSG}; Write(Var^{out}(P) \cap V_i, <_V), \nu_i, \phi_i)$$

La deuxième régle définit la lecture d'une entrée. La valuation doit être mise à jour et des messages de transfert de valeur pourront être nécessaires. La transition est labellisée par la variable d'entrée lue et sa valeur.

[Input]

$$(E_D^P)_i \vdash (\texttt{INPUT(x)};\omega_i, \nu_i, \phi_i) \xrightarrow{x?a} (\texttt{BCAST(x)};\omega_i, \nu_i[x \mapsto a], \phi_i)$$

$$\forall a \in \{\top, \bot\}$$

Les deux prochaines règles décrivent le traitement des messages de transfert de valeur. Les messages sont lus du canal de réception et la valuation locale est mise à jour. Dès lors, il est nécessaire d'examiner les WHENs portant sur la variable mise à jour puisque leurs conditions peut avoir changé de valeur. Les $ø_i$ et $ø'_i$ représentent le contenu du canal de communication respectivement avant et après la transition. Un message est caractérisé par un tuple (*x*,*v*,*s*) où *x* est la variable mise à jour, *v* sa nouvelle valeur, et *s* le processus d'origine du message.

[Traitement des messages]

$$(E_D^P)_i \vdash (\texttt{MSG};\omega_i, \nu_i, \phi_i) \xrightarrow{\tau} (Treat(W_{i/\tilde{x}}, <_W); \texttt{MSG};\omega_i, \nu_i[\widetilde{x_k} \mapsto v_k], \phi'_i)$$

$$\phi_i = (x_1, v_1, s_1) \cdots (x_{k-1}, v_{k-1}, s_{k-1}) \cdot (x_k, v_k, s_k) \cdot \phi''$$
$$\phi'_i = (x_1, v_1, s_1) \cdots (x_{k-1}, v_{k-1}, s_{k-1}) \cdot \phi''$$
$$\forall l \in [1..k-1] : s_l \neq s_k$$

[Fin du traitement des messages]

$$(E_D^P)_i \vdash (\texttt{MSG};\omega_i, \nu_i, \phi_i) \xrightarrow{\tau} (\omega_i, \nu_i, \phi_i)$$

La règle suivante correspond au traitement d'une assignation. La valuation est mise à jour et, si la variable apparaît tildée sur un autre site, sa nouvelle valeur doit être transférée. De plus, les WHENs qui porte sur cette variable doivent être examinés.

[Assignation]

$$(E_D^P)_i \vdash (x := e;\omega_i, \nu_i, \phi_i) \xrightarrow{\tau} \begin{cases} (\texttt{BCAST(x)};Treat(W_{i/x}, <_W);\omega_i, \nu_i[x \mapsto \nu_i(e)], \phi_i) & \text{if } x \in \widetilde{Var}(P) \\ (\omega_i, \nu_i[x \mapsto \nu_i(e)], \phi_i) & \text{if } x \in OldCond(P) \end{cases}$$

$$\text{où } W_{i/x} = \{w \in W_i \mid x \in Var(Cond(w))\}$$

La règle suivante correspond au traitement d'une instruction IF. Si la condition est évaluée à vrai (⊤), le code de la partie THEN est inséré à l'espace de travail. Sinon (⊥), c'est la partie d'instructions liée au ELSE qui est insérée. Celle-ci est vide dans le cas d'un IF sans code de ELSE.

[If]

$$(E_D^P)_i \vdash (\texttt{IF } e \texttt{ THEN } \omega_\top \texttt{ ELSE } \omega_\bot \texttt{ ENDIF};\omega_i, \nu_i, \phi_i) \xrightarrow{\tau} \begin{cases} (\omega_\top;\omega_i, \nu_i, \phi_i) & \text{if } \nu_i(e) = \top \\ (\omega_\bot;\omega_i, \nu_i, \phi_i) & \text{if } \nu_i(e) = \bot \end{cases}$$

La dernière règle locale correspond à l'écriture d'une variable de sortie. Une transition étiquetée par la variable de sortie et sa valeur est prise.

[Output]

$$(E_D^P)_i \vdash (\texttt{OUTPUT(x)};\omega_i, \nu_i, \phi_i) \xrightarrow{x!\nu_i(v)} (\omega_i, \nu_i, \phi_i)$$

Suivant ces règles sémantiques, étant donné un programme dSL et sa distribution, il est possible de définir une système à transitions étiquetées décrivant le comportement du programme.

Définition 9 (Sémantique distribuée d'un programme dSL) Etant donné un programme correct dSL *P* et une distribution $D = \{V_1,..,V_n\}$ de *P*, la sémantique distribuée de *P*, notée $[\![P]\!]_D$, peut être définie par un système à transitions étiquetées

$$(\mathcal{G}_D^P, G_D^0, (V(P) \times \{!, ?\} \times \{\top, \bot\}), \rightarrow)$$

où

- $G^0_D \equiv ((\omega_1, \nu_1, \varnothing_1), \dots , (\omega_v, \nu_n, \varnothing_n))$ avec $\forall\, i \in [1,n]$,
  - $\omega_i = \varepsilon$
  - $\nu_i(x) = \bot, \forall x \in (V_i \cup \widetilde{Var(P)} \cup OldCond(W_i))$
  - $\phi_i = \varepsilon$
- $\rightarrow$ est telle que $(G, a, G') \in \rightarrow$ si et seulement si $E_D^P \vdash G \xrightarrow{a} G'$ peut être dérivé de toute règle opérationnelle structurelle donnée précédemment.

♦

G^0_D est l'état initial du système, c'est-à-dire l'état dans lequel se trouve le système lorsque celui-ci est lancé. Tous les espaces de travail et les canaux de communication sont vides.

Chapitre III

Les Lego-Mindtsorms

Dans ce chapitre, nous introduirons les Lego-Mindstorms et décrirons l'environnement d'exécution. Ensuite, après avoir détaillé les différents langages de programmation possibles, nous nous pencherons sur le choix d'un de ces langages en fonction entre autres des communications infrarouges. Nous détaillerons pour finir quelles sont les étapes de la compilation vers les Lego-Mindstorms.

1. Introduction aux Lego-Mindstorms

Les Lego-Mindstorms [**Bau00-BGT00**] sont dérivés de recherches sur la robotique et sur son potentiel vis-à-vis de l'éducation à la science et à l'ingénierie. C'est un projet du *Massachusetts Institute of Technology*, la brique programmable MIT-PB qui est à l'origine de l'élément central des Lego-Mindstorms : la brique RCX (Robot Control Explorer) représentée à la figure 3.1.



Figure 3.1 : La brique RCX.

Une brique RCX est un petit système informatique complet avec un CPU, un affichage, de la mémoire et des périphériques. En effet, elle comporte un microcontrôleur Hitachi, ainsi que trois entrées identifiées par les chiffres 1, 2 et 3, trois sorties identifiées par les lettres A, B et C et un écran LCD. Quatre composants principaux peuvent être utilisés : des moteurs dont le sens et la vitesse de rotation peuvent être paramétrés et des capteurs de toucher, de lumière et de rotation. Ceux-ci sont détaillés à la figure 3.2. Les Lego-Mindstorms offrent d'autres possibilités plus ludiques telles que l'utilisation de petites lumières ou la production de sons.

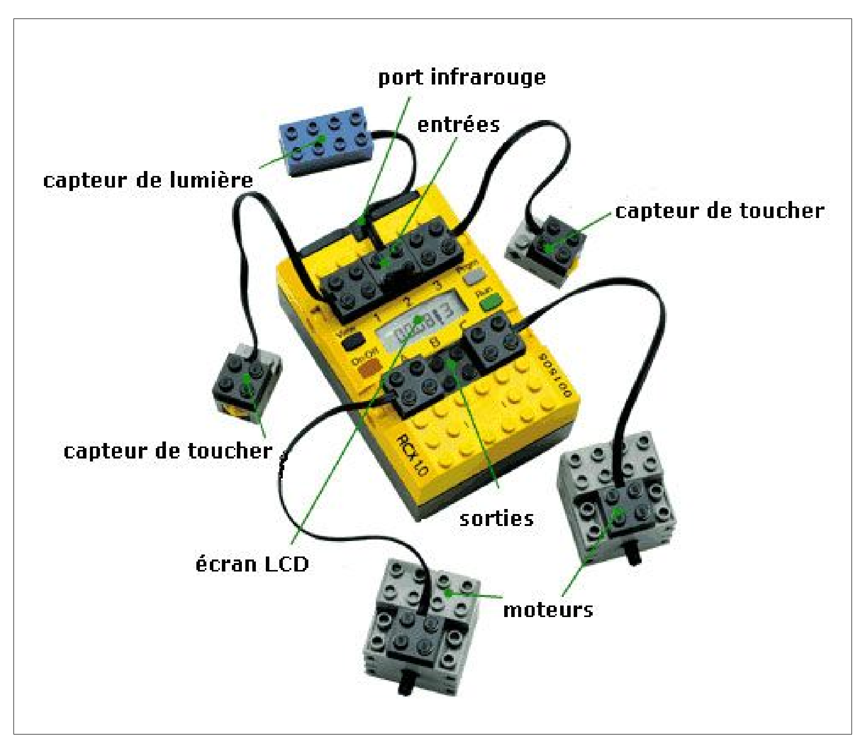


Figure 3.2 : La brique RCX et ses accessoires.

Les briques sont munies d'un port de communication infrarouge. Par l'intermédiaire de ce port, chaque brique RCX peut non seulement recevoir un programme à effectuer mais aussi communiquer avec un ordinateur, comme illustré à la figure 3.3, ou avec d'autres briques.

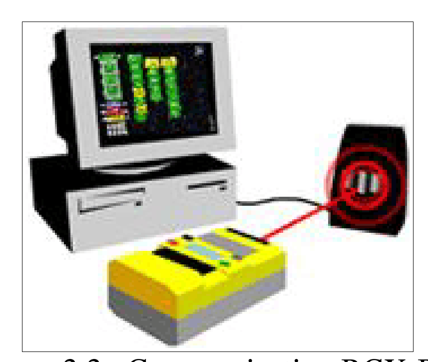


Figure 3.3 : Communication RCX-PC.

Les Lego-Mindstorms peuvent être dès lors utilisés comme laboratoire pour la conception et la vérification d'applications distribuées. Elles offrent en effet toutes les caractéristiques requises à savoir la possibilité d'avoir plusieurs unités distinctes capables d'exécuter un certain nombre d'instructions, d'interagir avec son environnement par l'intermédiaire des entrées et sorties ainsi que de communiquer avec les autres unités par l'intermédiaire de leur port infrarouge.

2. Définition de l'environnement Lego-Mindstorms

En vue de concevoir des applications distribuées sur différentes briques Lego-Mindstorms, il est important de détailler quelque peu les caractéristiques des composants principaux de celles-ci. En effet, la compréhension de l'architecture et du fonctionnement des RCX permet d'obtenir plus de fonctionnalité et de sûreté pour les systèmes créés.

La brique RCX utilise le microcontrôleur Hitachi H8/3292 [**HIT**] analogique 8 bits étendu avec 32kb de RAM externe. Il supporte un espace d'adressage de 16 bits et possède 16 registres de 8 bits (R0H, R0L, … R7H, R7L). Il y a deux registres de contrôle : le program counter (PC, 16 bits) et les registres de code des conditions (CCR, 8 bits). Le registre R7 est utilisé comme registre de stack pointer et renvoie vers le sommet du stack. Toutes les opérations sur le stack y accèdent en utilisant des mots de 2 bytes.

Le CPU dispose de 16 kb de mémoire ROM, contenant des softwares fournis par le fabriquant. La ROM contient un driver qui est lancé dès l'allumage de la brique et un ensemble de routines de bas niveau permettant de contrôler la RCX (on/off, affichage, son). Le driver ROM lancera le firmware, un software stocké dans la mémoire RAM externe. D'origine, il contient un interpréteur de code binaire qui ne laisse que 6kb de RAM aux autres programmes. Ceci aura une influence plus tard sur le choix du langage de programmation.

La RAM externe est référencée dans le range 0x8000-0xFFFF, une partie de cet espace d'adressage est réservée pour les entrées/sorties. D'autres parties de la mémoire sont réservées pour l'affichage de l'écran LCD, le contrôle du moteur, les registres shadow des ports I/O et le vecteur d'interruptions. La RAM est divisée en deux parties : noyau et utilisateur. La partie noyau concerne le firmware ou l'OS alternatif, la partie utilisateur correspond aux programmes et aux données.

Le microcontrôleur H8/3292 est en outre équipé d'un certain nombre de sous-systèmes. Une horloge 16 bits, les ports infrarouges, un convertisseur analogique/digital et enfin un module de timer 8 bits avec deux canaux. Ces derniers étant utilisés pour générer les signaux envoyés au haut-parleur et pour les communications infrarouges.

Deux briques RCX peuvent communiquer par l'intermédiaire de leurs ports infrarouges. Cette fonctionnalité est essentielle pour les systèmes que nous implémenterons. Les différentes briques devront par exemple s'échanger des valeurs de variables. L'utilisation du firmware standard limite les transferts à un byte de données. Cette limite peut être dépassée en remplaçant le firmware par un nouveau système d'exploitation. Le choix du langage utilisé pour la programmation des briques s'avère donc important.

3. Les différents systèmes d'exploitation et leurs langages

Hormis le langage Mindscript défini originellement par le software Lego, il existe différents langages et systèmes d'exploitation qui ont été développés au fil des années '90 pour programmer la brique RCX : Not Quit C, LegOS, brickOS, LejOS, Pbforth, etc. Nous allons à présent détailler les caractéristques principales de chacun de ces langages et systèmes d'exploitation.

Not Quit C [Bau00-BGT00]

Not Quit C est un langage de programmation qui a été développé par Dave Baum. Comme son nom l'indique, il est proche du C. En fait, il en reprend la syntaxe en y incorporant les commandes Mindstorms.

Les programmes binaires obtenus après compilation sont compatibles avec le firmware fourni par Lego. Dès lors, ce langage offre l'avantage de pouvoir utiliser simultanément des programmes Mindscript développés avec l'environnement graphique proposé par Lego et des programmes Not Quit C.

Cependant, il impose également de respecter les différentes contraintes imposées par le firmware comme par exemple le type des variables (uniquement des entiers) ou encore le nombre de celles-ci (au maximum 32 globales et 16 locales). Pour le problème qui nous motive ici, ce choix de langage est inapproprié. Il apparaît clairement que nous avons inévitablement intérêt à trouver un système d'exploitation alternatif qui ne nous impose pas ce type de contraintes.

LegOS [BGT00]

LegOS, développé par Markus Noga, a été le premier système d'exploitation alternatif à destination du RCX. LegOS est très complet : il est multitâche, dispose d'un allocateur dynamique de mémoire, permet le chargement dynamique de programmes grâce à l'interface infrarouge et gère l'ensemble des périphériques disponibles pour la RCX. LegOS est un système d'exploitation qui fournit des librairies permettant de remplacer le firmware de la brique par un OS plus puissant. Le code rédigé en C/C++, est compilé en un code binaire plus compacte que ceux obtenus avec LejOS.

Toutefois, même si LegOS offre un mécanisme de configuration statique, c'est un noyau monolithique, difficilement modifiable et peu évolutif.

BrickOS

C'est le successeur de LegOS. Il fournit un environnement de développement pour les RCX basé sur les outils gcc et g++ ainsi que les outils nécessaires pour télécharger les programmes. Conçu à l'origine pour Linux, il est également utilisable sous Windows.

LejOS [LFS02]

Une alternative au firmware serait de remplacer celui-ci par une Machine Virtuel Java (JVM). A l'instar de LegOS, cette solution offre de nombreux avantages par rapport à Not Quit C : nombre de variables illimité, utilisation de strings, etc.
LejOS reprend une partie du code de LegOS en essayant de le modulariser. De plus, LejOS intègre une micro-machine virtuelle Java et permet le chargement dynamique de classes Java à travers l'interface infrarouge du RCX. Il devient donc possible de programmer les Lego-Mindstorms directement en Java.

Toutefois, le noyau résultant est relativement lourd. L'espace nécessaire pour la JVM représente 9 des 32 Kb disponibles sur la RCX. Dès lors, cette solution semble être coûteuse.

Pbforth [BGT00]

A ne pas mettre entre toutes les mains. Pbforth n'est pas un langage de débutant. C'est la version RCX du langage Forth. Il est basé su une machine à pile. Son avantage est que l'interpréteur est dans la mémoire de la brique. Ainsi, il est possible de télécharger des programmes à partir de n'importe quelle machine capable d'accéder au port USB en mode ASCII.

En fait, Pbforth a trois caractéristiques principales :
- l'interactivité : les systèmes interactifs sont des systèmes recevant des commandes de l'utilisateur et répondant directement à celle-ci,
- l'interprétation : c'est-à-dire le fait que Pbforth soit un langage interprété -ce qui implique une certaine lenteur à l'exécution-,
- l'extensibilité : l'interpréteur peut être étendu à de nouveaux mots pour modifier la manière de gérer les données.

D'ors et déjà, nous pouvons sortir deux systèmes d'exploitation du lot : LejOS et LegOS (ou brickOS). Ces deux systèmes d'exploitation imposent chacun un langage particulier, l'un inspiré de Java et l'autre du langage C.

4. Les communications infrarouges

En vue de choisir un de ces systèmes d'exploitation et donc le langage destination de la compilation, il ne faut pas oublier qu'il sera nécessaire de gérer les communications inter-brique. Qui plus est de la manière la plus efficace et fiable possible. Dès lors, il est primordial d'examiner les différents protocoles de communications existants.

Le protocole de transport de données doit satisfaire certaines conditions :
1. unidirectionnel : Les données seront envoyées d'une RCX vers une autre sans aucune réponse nécessaire. Par là, nous voulons dire qu'il n'y aura pas de 'dialogue' entre deux RCX mais pas forcément que plusieurs messages ne seront pas échangés. En d'autres termes, quand un site envoie un message (typiquement un transfert de valeur ou un ordre d'exécution), il n'attend jamais une réponse de son destinataire. Mais en pratique, le protocole de communication pourra imposer l'échange de plusieurs messages d'acquittement ou de retransmission.
2. correction des données : les données doivent avant tout être correctes. En effet, il s'agit le plus souvent de transferts de valeur de variables.
3. fiabilité : les programmes qui sont implémentés requièrent une très grande fiabilité. En plus de satisfaire la fiabilité dans le cas de briques statiques, il faudra aussi envisager le cas de briques mobiles.
4. simplicité : il faut toujours garder à l'esprit que l'espace mémoire disponible est limité.

Lorsqu'on parle de protocole de transferts de données, nous pensons immédiatement aux deux protocoles de communication : UDP et TCP.

UDP assure la correction des données mais n'est pas fiable. Il est sans connexion contrairement à TCP qui établit une connexion entre les deux stations communicantes. TCP, par contre, est un protocole fiable orienté connexion. La connexion établie entre deux stations - appelées sockets - est bidirectionnelle. Il assure également la correction des données mais nécessite plus de ressources mémoire pour être mis en œuvre.

Le protocole LNP est un protocole qui a été développé pour LegOS/brickOS. Il est UDP-like. Il permet l'envoi de messages point-à-point, c'est-à-dire définis entre deux RCX et deux seulement. Il permet en outre l'envoi de messages de plusieurs bytes en les incluant dans un paquet. Le paquet est généralement constitué d'un byte de début, du ou des bytes de messages et d'un byte de fin. Il existe 2 types de messages générés : les messages adressés (*addressing message*), c'est-à-dire pour lesquels un destinataire unique est spécifié, et les messages dédiés à toutes les briques du réseau, dits d'intégrité (*integrity message*).

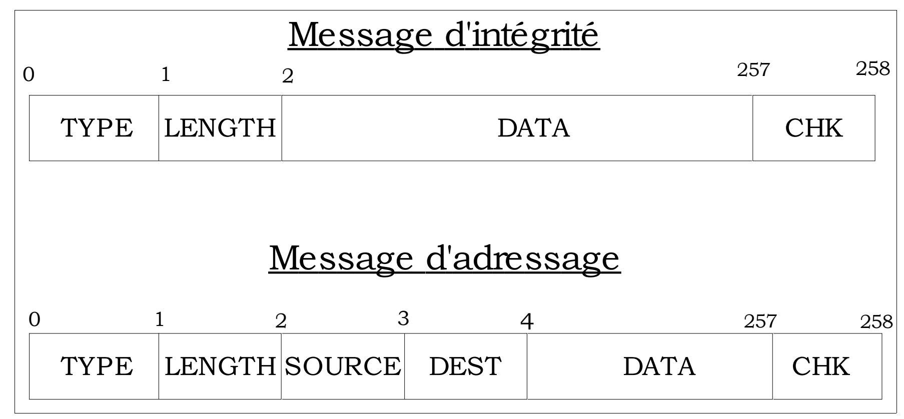


Figure 3.4 : Formats des messages lnp.

La structure des paquets du protocole LNP est reprise à la figure 3.4. Un champ de type permet de spécifier le type du message, le champ *LENGTH*, sa longueur. Dans le cas d'un message adressé, deux champs de soucre et de destinataire sont nécessaires. Le champ *CHK* (*checksum*) permet de tester la correction des données.

En ce qui concerne LejOS, certaines classes Java ont été spécialement définies pour les communications RCX. LNP est disponible grâce à **josx.rcxcomm.LNP**. Il est possible d'envoyer et de recevoir des messages d'intégrité ou d'adressage mais ce n'est pas fiable.

À titre d'information complémentaire, signalons qu'il existe actuellement des langages en cours de développement pour la communication inter-robots tels que Combot. Il sera intéressant de suivre l'évolution de ceux-ci dans le futur pour l'amélioration du comportement et des performances des robots.

5. Choix d'un système d'exploitation

D'une part, LejOS fournit

- un OS alternatif au firmware standard,
- une machine virtuelle,
- un protocole de communication,
- un ensemble de librairies,
- une grande portabilité.

Au vu de la portabilité offerte par le langage Java, LejOS peut s'avérer être un choix judicieux. En effet, le travail effectué pourra être facilement transporter à d'autres situations sur d'autres supports utilisant une machine virtuelle Java JVM. Mais la place mémoire disponible pour les programmes utilisateurs et les ressources utilisées par le système d'exploitation sont malheureusement limitées.

D'autre part, LegOS est un choix de système d'exploitation intéressant dans le problème qui nous concerne à savoir la génération de code de systèmes distribués.
Il fournit

- un OS puissant alternatif au firmware standard,
- d'avantage de place en mémoire disponible,
- un protocole de communication,
- un ensemble de librairies,
- la possibilité de faire du multitâche sans préemption,
- une gestion dynamique de la mémoire …

LegOS a donc été choisi comme langage cible de la génération de code. En ce qui concerne les communications, le protocole LNP, originellement développé pour LegOS, fonctionne très bien pour les transferts de données et offre de bons résultats dans un environnement suffisamment favorable. Cela dit, nous serons amené à envisager l'utilisation d'un protocole fiable de communication utilisant la couche LNP pour l'envoi des messages et assurant la fiabilité des communications.

6. La compilation vers LegOS

Le compilateur sera donc chargé de créer un programme intermédiaire en C qui sera ensuite compilé pour LegOS par un compilateur existant appelé *cross-compiler* ou compilateur croisé. Le parcours d'un programme dSL est repris par la figure 3.5. Une partie du compilateur, le distributeur, sera chargé de répartir les variables et les instructions sur les différents sites composant le système (cf. *Chapitre II Le langage dSL)*. Pour chaque site, le compilateur est chargé de créer deux fichiers (*.c et *.h) qui constitueront le programme utilisateur qui sera chargé sur chacune des briques. Les programmes utilisateur sont compilés dans un format relocatable avec l'extension .lx. Ils sont ensuite chargés dans la brique grâce à la fonction dll qui les transfère par l'intermédiaire des ports infra rouges du PC et des briques.

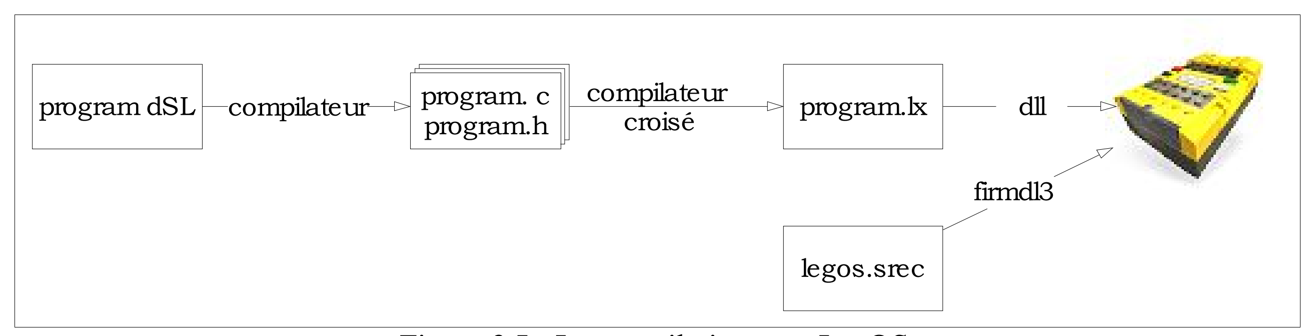


Figure 3.5 : La compilation vers LegOS.

En ce qui concerne le noyau, la directory de boot contient deux fichiers : LegOS.srec (image du noyau) et LegOS.lds (script du linker généré lors de la compilation). Le système d'exploitation doit être chargé sur chaque brique avant d'y transférer le programme utilisateur. Ceci s'effectue grâce à firmdl3. Plus d'informations sur le noyau LegOS peuvent être trouvée dans [**Nie00**].

La figure 3.6 reprend l'architecture d'une brique RCX après chargement du système d'exploitation et du ou des programmes utilisateur. Un programme utilisateur utilise les primitives du système d'exploitation qui utilise la couche système (interruptions, ...) et de la couche hardware (accès aux entrées et sorties, ...).

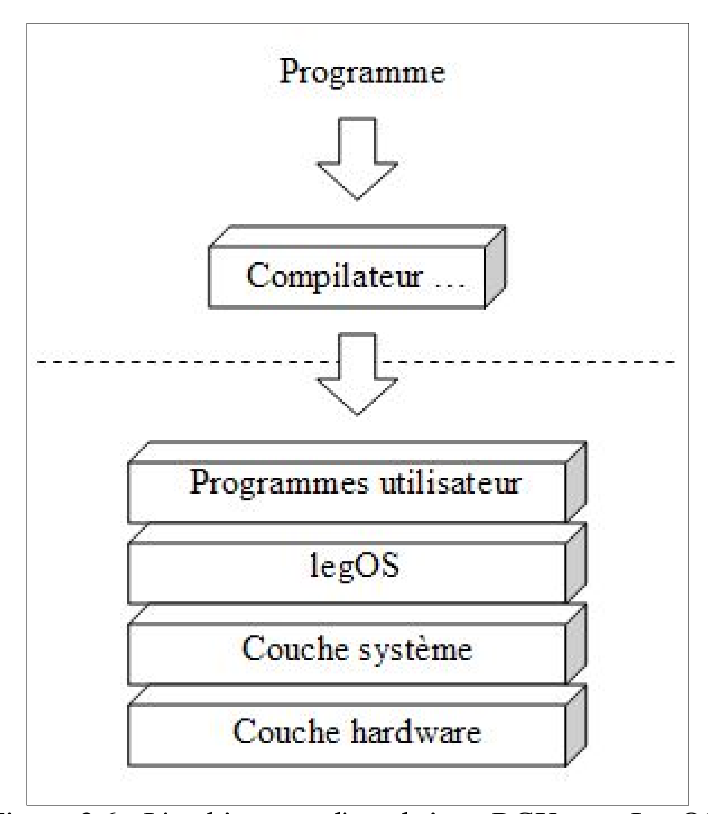


Figure 3.6 : L'architecture d'une brique RCX sous LegOS.

La conception de programmes destinés aux briques Lego n’est pas sans danger. Une erreur de manipulation de données (par exemple écrire une valeur dans un mauvais emplacement en mémoire) peut crasher l’application et rendre la brique RCX inactive. Pas de panique ! Il suffit d’enlever les batteries de la RCX et de recharger le système d’exploitation et l’application préalablement corrigée.

Signalons qu'un petit manuel d'utilisation de LegOS a été ajouté à l'*Annexe C*.

Chapitre IV

Compilation et distribution

Dans ce chapitre, nous commencerons par introduire les différentes étapes de la compilation. Ensuite, nous détaillerons la structure du compilateur et les rôles de chacun de ces constituants, notamment ceux du distributeur.

1. Le chemin de compilation

La figure 4.1 reprend l'ensemble des étapes que le programme du système va traverser. Comme nous l’avons dit précédemment, le code est généré par le compilateur en langage C. Il sera ensuite compilé par un compilateur croisé gcc vers le format LegOS, plus précisément vers un format relocatable .lx. Les fichiers .lx sont finalement chargés sur les briques RCX (cf. *III.6 La compilation vers LegOS*). La version LegOS que nous avons utilisée (0.2.6) et le *cross-compiler* sont disponibles sur le net (http://legos.souceforge.net). Trois fichiers de librairies de méthodes prédéfinies seront attachés au code généré. Il s’agit du fichier des méthodes nécessaires pour la mise en œuvre des systèmes générés (dsl_vm.h) et des fichiers du protocole de communication fiable (ab.c et ab.h). Le code de ces librairies est disponible respectivement en *Annexes D* et *E*. Remarquons que le compilateur génèrera deux fichiers (.c et .h) par site contenant entre autres l'ensemble des déclarations des variables et des instructions assignées statiquement à ce site.

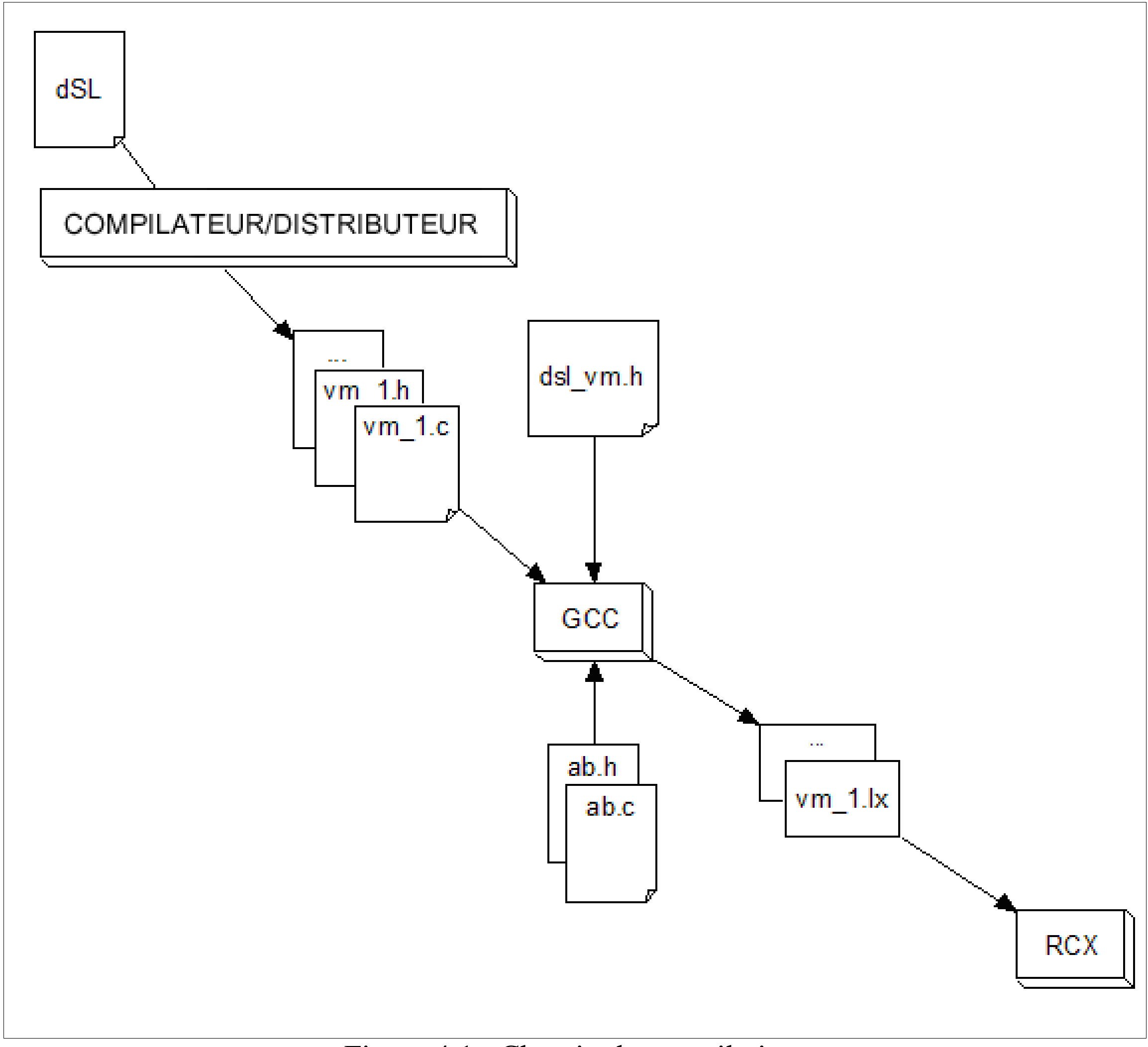


Figure 4.1 : Chemin de compilation.

2. La structure du compilateur

Un compilateur est un programme chargé de traduire un programme source en un programme cible équivalent. Dans le cas présent, le langage d'origine est dSL, le langage cible est C. Le compilateur est constitué de trois parties : la partie frontale ou *frontend*, le distributeur et la partie finale ou *backend*. Le *frontend* réalise tout ce qui est analyse du code du programme source, le distributeur distribue celui-ci sur les différents sites et le *backend* génère le code cible des sites.

2.1. Le *frontend*

L'analyse d'un programme consiste à identifier tous ses constituants lexicaux (mots-clés, opérateurs, nombres, identificateurs de variable, de méthode, etc.) et à en créer une représentation intermédiaire sous forme de blocs de base et de graphe de flot de contrôle. Un bloc de base est une suite d'instructions consécutives telle que le contrôle rentre en son début et ne peut être ni arrêté, ni en sortir avant la fin de celui-ci, c'est-à-dire avant d'avoir exécuté la dernière de ses instructions. Un bloc de base est constitué soit d'une suite d'instructions d'assignation, soit d'un saut vers un autre bloc de base (conditionnel ou inconditionnel, avec ou sans paramètre), soit un appel vers un autre graphe. Un graphe de flot de contrôle (ou *control flow graph*) représente la structure du programme,

c'est-à-dire les relations entre les blocs de base constituant ce programme. Il s'agit d'un graphe orienté avec éventuellement des cycles. Les noeuds sont les blocs de base. Un arc orienté joint un bloc de base à un autre si, lors de l'exécution du programme, le contrôle peut passer de la dernière instruction du premier bloc de base à la première instruction du deuxième. Remarquons que parfois certains blocs de base intermédiaires vides sont ajoutés par le parser. C'est le cas par exemple dans le graphe de flot de contrôle d'une instruction conditionnelle IF. La figure 4.2 reprend un graphe de flot de contrôle représentant un IF imbriqué dans un WHILE. Tant que la condition du WHILE sera vraie, le contrôle passera par les blocs de base de gauche, c'est-à-dire ceux du IF. Si la condition du IF est vraie, le contrôle va passer par le ou les blocs de base de gauche; sinon, ceux de droite. Lorsque la condition du WHILE est fausse, le contrôle sort de la construction WHILE et passe à l'instruction suivante.

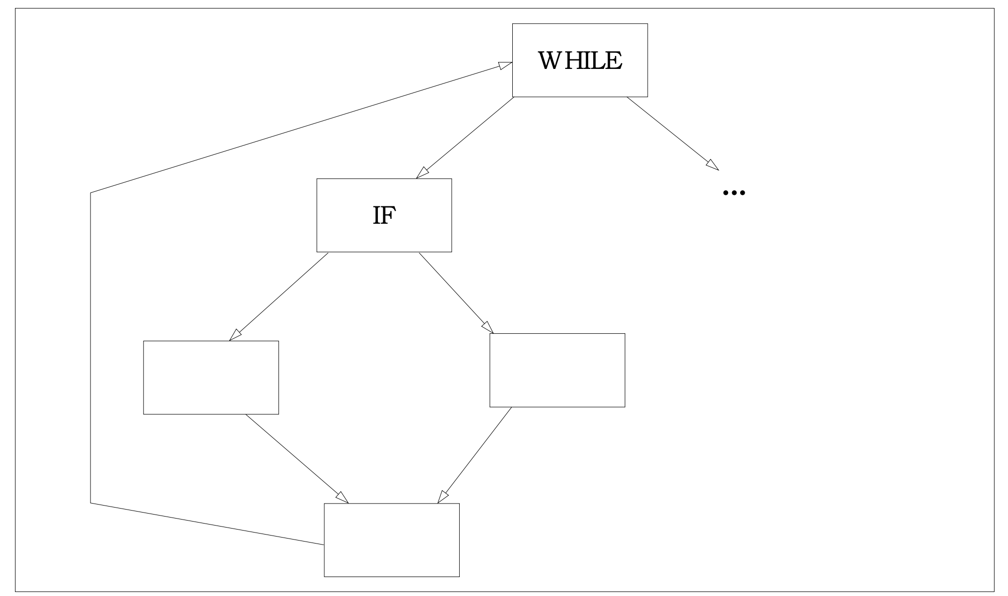


Figure 4.2 : Exemple de graphe de contrôle de flot d'un if imbriqué dans un while.

Ces blocs de base vides sont enlevés par un algorithme d'optimisation pendant la compilation (cf. **[Dew02]**). La figure 4.3 présente le graphe de contrôle de flot de la figure 4.2 après optimisation.

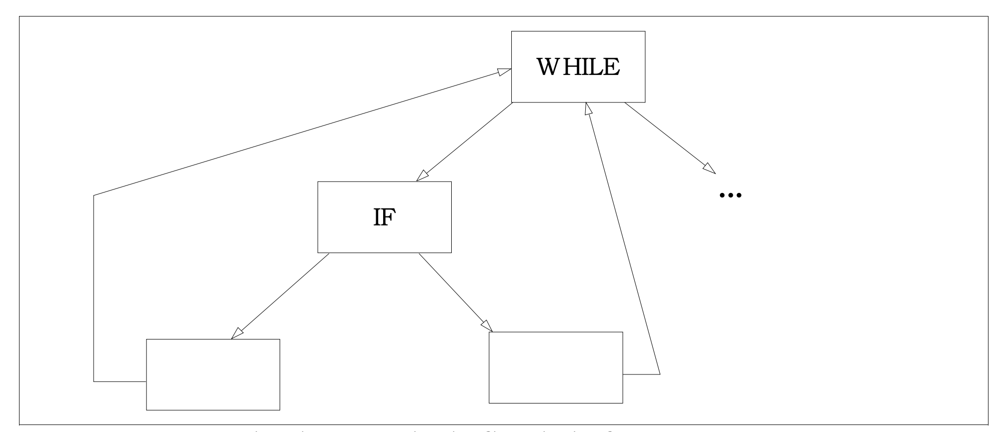


Figure 4.3 : Graphe de contrôle de flot de la figure 4.2 après optimisation.

En plus de la conception du graphe de contrôle de flot, le *frontend* associe à chaque variable, méthode ou séquence, un certain nombre d'informations nécessaires, tels qu'un identificateur unique (entier de 16 bits), le type ou les paramètres éventuels :

1. Variable : le type est celui de sa déclaration dans le programme dSL. A chaque utilisation de la variable dans une instruction, la correspondance de type est vérifiée. Les paramètres sont nécessaires en cas de variable d'entrée ou de sortie. Il s'agit de sa sorte (entrée ou sortie) et des trois nombres qui identifient le type et l'emplacement du dispositif (cf. *II.9.7. Les sites et les entrées/sorties*).

2. Méthode : Les méthodes n'ont pas de valeur de retour. Il n'y a donc pas de type associé à une méthode, juste ses éventuels paramètres.

3. Séquence : Il n'y a pas de type associé à une séquence, juste ses éventuels paramètres.

2.2. Le distributeur

Comme son nom l'indique, sa tâche essentielle est de distribuer statiquement les instructions sur les différents sites. Pour cela, il va utiliser les résultats de l'analyse du code. Une technique de distribution consiste à colorier le graphe de contrôle de flot des instructions. Une solution optimale de coloriage d'un graphe de flot basée sur une fonction de coût est présentée dans [**Dew02**]. Le coût est déterminé grâce à une grammaire attribuée qui calcule le nombre moyen de migrations qui se produiront lors de l'exécution. Cette technique minimise cette fonction en assignant à chaque instruction une couleur, c'est-à-dire un site. De plus, elle respecte le coloriage initial, défini sur base de la distribution des variables spécifiée dans le programme dSL et de la contrainte atomique (cf. *II.6 Code atomique et séquentiel*). Cette technique de coloriage est optimale mais requiert la résolution d'un problème NP-complet. Une technique de réduction du graphe de flot et l'utilisation d'une heuristique locale permettent d'obtenir, en un temps raisonnable, une solution acceptable.

En fonction de la distribution, le distributeur réalise aussi l'analyse des variables actives. Pour un bloc de base donné, une variable du bloc est dite active si elle est utilisée par une instruction d'un bloc situé plus loin dans le chemin d'exécution sans que celle-ci soit redéfinie sur ce chemin. L'analyse des variables actives nécessite de déterminer quatre ensembles pour chaque bloc de base B, à savoir :

1. Def[B] : l'ensemble des variables qui reçoivent une valeur dans B avant d'être utilisée dans B,
2. Use[B] : les variables qui peuvent être utilisées dans B avant toute redéfinition,
3. In[B] : l'ensemble des variables actives à l'entrée du bloc,
4. Out[B] : l'ensemble des variables actives à la sortie du bloc.

Pour chaque bloc B, Def[B] et Use[B] peuvent être déterminés directement sur base du contenu du bloc. In[B] et Out[B] sont calculés suivant les formules de la figure 4.4 en réalisant une analyse en arrière sur le graphe de flot des données du programme (cf. **[ASU00]**).

```
in[B]= use[B] ∪ ( out[B] - def[B] )
out[B] = ∪ { in[s] | s successeur de B }
```

Figure 4.4 : Formules de In[B] et Out[B].

Sur base du coloriage et de l'analyse des variables actives, le distributeur va insérer des points de migration dans le code et plus particulièrement à l'intérieur des séquences. Un point de migration entre deux instructions consécutives d'une même séquence est nécessaire lorsque les instructions sont localisées sur des sites différents. L'analyse des variables actives permet de déterminer le sous-ensemble des variables du contexte local qui doivent être transmises, c'est-à-dire les variables dont la valeur est utilisée plus loin par une instruction située sur un autre site avant laquelle les variables ne sont pas assignées. Pour rappel, les événements et les méthodes sont tenus de respecter la contrainte atomique. C'est pourquoi les points de migration ne sont introduits qu'au sein des séquences.

2.3. Le *backend*

Il est chargé de générer le code cible du programme sur base des analyses réalisées par les deux autres parties du compilateur. Pour chaque site du système, il va donc créer deux fichiers (`.c` et `.h`) qui contiendront le code des composants du système. Ce code doit être correct et efficace. Nous avons veillé également à ce que le code généré soit de bonne qualité, c'est-à-dire structuré, lisible et aisément compréhensible pour le programmeur, avec par exemple l'ajout de commentaires d'explication. La génération automatique du code est l'objet du chapitre suivant.

Chapitre V

Génération automatique de code LegOS pour Lego-Mindstorms

Dans ce chapitre, nous présenterons l'apport que nous avons fait au compilateur existant, tant dans la réflexion que dans l'implémentation. Nous étudierons la manière dont le code C est généré par le compilateur et nous nous efforcerons de décrire les principaux mécanismes employés pour mettre en œuvre les constructions dSL en langage C. Nous illustrerons également la génération de code par une série d'exemples. Pour finir, nous présenterons également le protocole de communication fiable que nous avons implémenté.

1. Les tâches

En vue de générer du code ayant un comportement équivalent à celui du programme dSL source, nous serons amené à gérer des tâches, et plus précisément à les créer, les paramétrer, les mettre en attente et les réveiller. Dans la suite, nous appelerons tâche, un bout de programme contenu en mémoire et qui s'exécute sans perturber le fonctionnement des autres tâches. Outre le code de ses instructions, elle possède un contexte local, c'est-à-dire ses variables locales sous forme de pile et un *program counter* qui précise où la tâche en est dans son exécution. LegOS est un système multitâche préemptif dans lequel chaque tâche possède une priorité. En fonction de ces priorités, le système d'exploitation va donner le contrôle à la ou les tâches les plus prioritaires. A tout moment, il peut reprendre le contrôle à une tâche pour le donner à une nouvelle tâche plus prioritaire par exemple.

La priorité d'une tâche est définie lors de sa création. Il faut également préciser les instructions que la tâche exécutera, ses paramètres et la taille de la mémoire qui lui sera accordée pour stocker son contexte. En legOS, les priorités vont de 1 à 20. Les instructions sont spécifiées par l'intermédiaire d'une fonction. Le code de la fonction correspond aux instructions que la tâche exécutera. Les paramètres de la tâche sont les paramètres de cette fonction. Ils sont passés sous forme d'un vecteur de chaînes de caractères. En ce qui concerne la mémoire d'une tâche, nous utiliserons la moitié de la variable legOS `DEFAULT_STACK_SIZE` qui équivaut à 512 bytes. La figure 5.1 reprend la méthode qui permet de créer une tâche. Cette méthode renvoie le *pid* de la tâche créée, c'est-à-dire son identificateur au sein du système d'exploitation. Il s'agit en réalité du pointeur vers le stack de la tâche. Pour toutes les tâches créées, cet identificateur sera conservé dans une variable particulière. Nous verrons pourquoi au paragraphe *3.3 le traitement des messages*.

```
pid_t execi(fonction, argc, *argv[], priorité, taille_du_stack)
```

Figure 5.1 : Fonction de création d'une tâche LegOS.

Nous avons précédemment intoduit le concept d'automate et plus précisément la boucle *Input-Process-Output* (cf. *II.1. Introduction au langage SL*). Cette boucle doit être exécutée en permanence par chaque site. Elle fera donc l'objet d'une tâche permanente sur chaque site. La partie *Process* concerne la gestion des événements et des messages. Lors du traitement des messages, elle sera parfois amenée à suspendre son exécution pour exécuter des méthodes ou des séquences distribuées sur son site. Les séquences seront traduites par des tâches supplémentaires puisqu'il faut conserver son contexte local. En effet, lorsqu'une séquence dont les instructions sont distribuées sur plusieurs sites est exécutée, le contrôle va devoir migrer de site en site. Par exemple, la séquence présentée dans la figure 5.2 doit suspendre son exécution sur le site 1 en attendant de reprendre le contrôle après que les instructions de la deuxième partie de la séquence distribuée sur le site 2 aient été exécutées. Chaque séquence est traduite par une tâche dont l'exécution sera suspendue aux points de migration en attendant d'être réveillée pour continuer son exécution. Remarquons qu'il n'y a pas de concurrence dans l'exécution des tâches pour un site donné. Ce qui signifie qu'il faut veiller à n'avoir, à tout moment, qu'une et une seule tâche active.

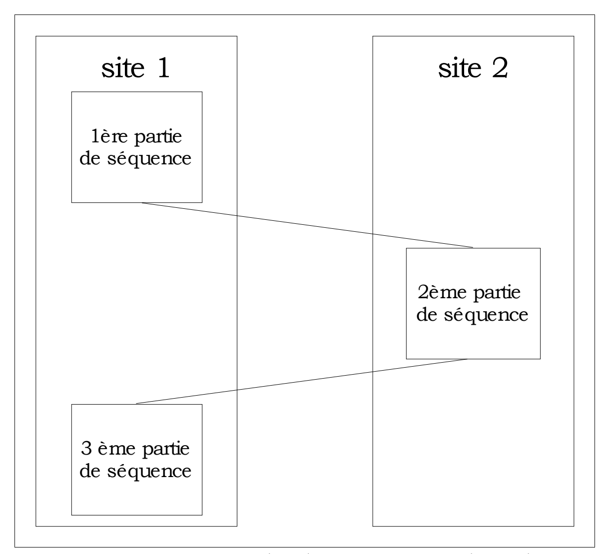


Figure 5.2 : Exemple de séquence distribuée.

Nous devons donc pouvoir 1. suspendre l'exécution d'une tâche en cours et 2. passer le contrôle à une autre tâche en particulier. De plus, lorsque l'exécution d'une tâche assignée à une séquence s'arrête, la tâche doit rendre le contrôle à la tâche qui lui a donné la main.

Afin d'implémenter un tel comportement, trois solutions auraient été envisageables. La première consiste à modifier le scheduling du système d'exploitation par l'ajout de nouvelles primitives tels que *start_until_suspend*() et *suspend*() qui permettraient de suspendre l'exécution d'une tâche en restaurant la tâche qui lui a donné le contrôle. En plus de ses nouvelles primitives, il faudrait modifier le comportement du système d'exploitation en supprimant le caractère multitâche préemptif du noyau, c'est-à-dire tout ce qui concerne les interruptions de l'horloge, etc. Après compilation du noyau, nous obtiendrions un nouveau noyau modifié que nous devrions utiliser pour exécuter nos applications. Cela diminuerait la portabilité des applications générées par notre compilateur puisqu'il serait impératif d'utiliser le noyau modifié pour exécuter les applications. Modifier le noyau n'est donc pas une bonne idée. Il est préférable de laisser le noyau tel quel et d'utiliser ses primitives pour simuler le scheduling.

Une solution pourrait être alors de jongler avec les sections critiques en utilisant les sémaphores qui permettent de mettre en attente ou de réveiller une tâche. Cependant, rendre le contrôle à la tâche qui nous a rendu la main s'avère très complexe avec le seul usage des sémaphores.

Pour qu'une tâche rende le contrôle à la tâche qui l'a lancée, nous avons pensé à un stack de pid de contrôle sur lequel la tâche qui compte donner le contrôle à une autre, stocke son pid pour un retour de contrôle ultérieur. Ensuite, elle passe le contrôle à la tâche à lancer et se met en attente. Lorsque la tâche en cours d'exécution va vouloir rendre le contrôle, elle consultera le pid au sommet du stack de contrôle pour pouvoir réveiller la tâche qui lui a transmis le contrôle. Le passage de contrôle entre les tâches va se faire via un jeton de contrôle. En pratique, ce jeton consiste en une variable globale (« **current_thread_id** ») contenant l'identificateur de la tâche autorisée à s'exécuter, c'est-à-dire la tâche qui a obtenu ou qui doit obtenir le contrôle. Lorsqu'une tâche est mise en attente, elle attend en fait que le jeton soit pour elle, c'est-à-dire que la variable globale contienne son propre identificateur (accessible grâce au mot-clé LegOS « **cpid** »). La mise en attente d'une tâche est possible, en LegOS, par l'intermédiaire d'un événement (appelé par la suite événement LegOS). Cela consiste à suspendre l'exécution d'une tâche dans l'attente qu'un certain événement se produise. La figure 5.3 précise la syntaxe de la définition d'un événement LegOS où *id* est l'identificateur de l'événement et *paramètres*, les paramètres de l'événement. Le *code* d'un événement est constitué d'une suite d'instructions avec éventuellement des déclarations de variables locales et se termine par une instruction de retour portant sur une expression booléenne. Cette expression constitue la condition qui définit l'événement. En pratique, le système d'exploitation va régulièrement vérifier les conditions des événements. Si une tâche est en attente sur une condition et que celle-çi devient vraie, la tâche change son status de *sleeping* (endormie) à *waiting for CPU* (en attente du processeur). La figure 5.4 précise la méthode qui permet de mettre en attente une tâche sur un événement.

```
wake_up_t id (paramètres) {
    code
    return expression_booléenne ;
}
```

Figure 5.3 : Syntaxe d'un événement LegOS.

```
wait_event (wake_up_t(* wakeup)(wakeup_t), wakeup_t data)
```

Figure 5.4 : Syntaxe de méthode de mise en attente d'une tâche LegOS.

Nous avons donc défini l'évenement LegOS *getControl*(pid_t *id*) qui teste si le jeton est égal à *id*, l'identificateur de tâche fourni en paramètre. Dès lors, lorsque une tâche veut donner le contrôle à une tâche de séquence, elle stocke son pid sur le stack de contrôle, elle met à jour le jeton et elle se met en attente grâce à l'instruction « **wait_event(&getControl, myPid)** », où *myPid* est son propre pid. Une tâche de séquence dont l'exécution est terminée (fin de la séquence ou migration) passe obligatoirement le contrôle à la tâche qui le lui a donné. Elle met à jour le jeton avec le sommet du stack de contrôle et se met en attente de la même manière.

En pratique, la mise en attente et le réveil des tâches de séquence se fait via deux fonctions, respectivement *startwait*() et *wakeup*(pid_t * *id*) où *id* est l'identificateur de la tâche à réveiller, présentées dans la figure 5.5 et définies dans le fichier dsl_vm.h .

```
void startwait() {
    if (!StackIsEmpty()) {
        current_thread_id = popStack();
    }
    wait_event(&getControl,(unsigned int)cpid);
}

void wakeup(pid_t * id) {
    pushStack((pid_t*)cpid);
    current_thread_id = id;
    wait_event(&getControl, (unsigned int)cpid);
}
```

Figure 5.5 : Fonctions C de mise en attente et de réveil de tâche LegOS.

2. La tâche *Input-Process-Output*

Nous avons vu dans *II.1 Introduction au langage SL* que chacun des sites constituant le système devra exécuter continuellement la boucle *Input-Process-Output* qui fera donc l'objet d'une tâche par site, créée dès l'initialisation du système. La figure 5.6 reprend le code de la fonction contenant les instructions de la tâche principale. Cette fonction fait partie des méthodes prédéfinies du fichier dsl_vm.h dont le code est repris en *Annexe D*.

```
int start() {
     while(true) {
        sampleInputs();
        handleWhens();
        handleMsg();
        writeOutputs();
     }
     return 0;
}
```

Figure 5.6 : Fonction de la tâche de la machine virtuelle.

sampleInputs() est la fonction chargée de la lecture des entrées du site. *handleWhens()* est la fonction chargée du traitement des WHEN distribués sur le site. *handleMsg()* est la fonction chargée de la gestion des messages reçus par le site. Et finalement, *writeOutputs()* est la fonction chargée de la mise à jour des dispositifs de sortie du système. Le contenu des fonctions *sampleInputs()* et *writeOutputs()* est détaillé dans le paragraphe *7. Les entrées/sorties*. La fonction *handleWhens()* est détaillée dans le paragraphe *12. Les événements*. Le contenu de *handleMsg()* est explicité dans le paragraphe suivant au point *3.3. Le traitement des messages*.

La tâche *Input-Process-Output* est créée dans la fonction d'intialisation du site, c'est-à-dire la fonction *main*() du fichier .c du site. Une priorité inférieure à celle des tâches des séquences lui est accordée.

3. Les messages

3.1. Les types de messages

Etant donné la nature distribuée du système, des messages seront échangés entre les sites et cela pour diverses raisons :

1. Lancement de méthodes et de séquences sur des sites distants : Dans le cas d'une instruction de lancement d'une méthode ou d'une séquence (cf. *II.8.8.f Opérateurs de lancement « LAUNCH » et « <- »*), le site de distribution de la méthode ou de la première instruction de la séquence peut être différent de celui sur lequel l'instruction de lancement est distribuée. L'exécution de l'instruction nécessite un message entre ces deux sites. Il sera parfois nécessaire de transmettre les valeurs des paramètres de la méthode ou de la séquence.
2. Points de migration : Lorsque les instructions d'une séquence sont localisées sur des sites différents, le compilateur génère des points de migration du contrôle (cf. *2.2. Le distributeur*). Ces points de migration sont traduits par des messages de demande d'exécution d'une partie de séquence. Rappelons qu'il sera éventuellement nécessaire de transmettre une partie du contexte local.
3. Variables tildées : Lorsqu'une variable tildée est assignée, sa nouvelle valeur doit être transmise à tous les sites sur lesquels elle est utilisée.

Il existe donc trois types de messages :

1. Les messages de demande d'exécution d'une méthode,
2. Les messages de demande d'exécution d'une partie de séquence (le lancement d'une séquence est considéré comme la demande d'exécution de la première partie de la séquence),
3. Les messages de transfert de valeur d'une variable tildée.

Remarquons que selon le type de système considéré, il pourrait être nécessaire d'effectuer une synchronisation d'horloge entre certains ou tous les sites composant le système. Une synchronisation implique un certain nombre de messages échangés entre ces sites. Nous ne considèrerons pas ces messages.

3.2. Les fonctions de conception des messages

À chaque type de message est associée une fonction prédéfinie de conception et d'envoi du message. Ces fonctions sont définies dans la librairie `dsl_vm.h` (cf. *Annexe D*). À chaque fois qu'un message est nécessaire, une instruction d'appel à la fonction correspondante est générée dans le code du programme cible. Selon le type du message, celui-ci contiendra différents paramètres. Les formats de chacun des messages sont repris en *Annexe F*.

Dans le cas d'une demande d'exécution d'une méthode, le message devra contenir l'identificateur de la méthode ainsi que les valeurs des paramètres de celle-ci. L'ordre de ces paramètres est fixé par la définition de la méthode dans le programme dSL. Il n'est donc pas nécessaire de transmettre les identificateurs des variables passées dans le message. La syntaxe de la fonction de création et d'envoi du message est reprise dans la figure 5.7 où *id* est l'identificateur de la méthode, *site* est l'identificateur du site de distribution de la méthode, c'est-à-dire la destination du message et *n_para* contient le nombre de paramètres à transmettre à la méthode. Ces paramètres sont transmis sous forme d'une liste d'arguments de longueur variable (« **va_list** »). Une va_list est un type C qui va représenter chaque argument passé à une fonction à tour de rôle. En résumé, il faudra accéder *n_para* fois à une variable de ce type pour accéder aux *n_para* paramètres transmis à la place des « ... » dans la définition de la fonction.

void LAUNCH_ID_COLOR (int *site*, int *id*, int *n_para*, ...)

Figure 5.7 : Fonction d'envoi d'une demande d'exécution d'une méthode.

Dans le cas d'une demande d'exécution d'une partie de séquence, le message doit contenir l'identificateur de la séquence, l'identificateur de la partie et les valeurs des paramètres de la séquence ou des variables actives du contexte (cf. *2.2. Le distributeur*). Il sera nécessaire de transmettre les identificateurs des variables passées dans le message. Nous reviendrons plus loin sur ce point lorsque nous parlerons du traitement des messages. La fonction de création et d'envoi de ce type de message est détaillée à la figure 5.8 où *site*, *id* et *séquence* sont respectivement les identificateurs du site, de la partie de séquence et de la séquence. Ensuite, viennent les paramètres passés sous forme d'une va_list.

```
void HOP(int site, int séquence, int id, int n_para, ...)
```

Figure 5.8 : Fonction d'envoi d'une demande d'exécution d'une partie de séquence.

Dans le cas d'un transfert de valeur de variable tildée, le message doit contenir l'identificateur de la variable et sa nouvelle valeur. L'identificateur est nécessaire aussi car il peut y avoir plusieurs variables tildées utilisées sur un site. La fonction de création et d'envoi du message est reprise à la figure 5.9 où *id* et *val* sont respectivement l'identificateur et la valeur de la variable et *site*, le site de destination du message.

```
void SEND_ID_WARN (int site, int id, int val)
```

Figure 5.9 : Fonction d'envoi d'un transfert de valeur d'une variable tildée.

3.3. Le traitement des messages

Les messages reçus par un site sont stockés dans l'attente de leur traitement effectué par *handleMsg*() (cf. *2. La tâche Input-Process-Output*). Celui-ci est différent selon le type de messages :

1. Dans le cas d'une demande d'exécution d'une méthode, l'identificateur de la méthode et les paramètres éventuels sont transmis à la fonction *handleExec*(int *id*, char* *args*). Le code d'*handleExec*(), à savoir les appels aux méthodes du site selon l'identificateur transmis, est généré par le compilateur dans le fichier .c du site. La manière dont les paramètres sont transmis à la méthode est détaillée dans le paragraphe *10. Les méthodes*.
2. Dans le cas d'une demande d'exécution d'une partie de séquence, la tâche de la séquence doit être réveillée. La tâche courante, c'est-à-dire la tâche *Input-Process-Output*, est endormie dans l'attente de la fin de l'exécution de la partie de séquence à exécuter. La manière dont les séquences sont gérées et dont les paramètres sont transmis à la séquence est détaillée dans le paragraphe *12. Les séquences*.
3. Dans le cas d'un transfert de valeur d'une variable tildée, l'identificateur et la valeur de la variable sont transmis à la fonction *handleWarn*(int *id*, int *val*) dont le code est généré par le compilateur tel qu'en fonction de l'identificateur, la variable correspondante est assignée à la valeur transmise (cf. *10.La gestion des tildes*).

4. Les identificateurs

Le nom des variables dans le programme source dSL est conservé dans le programme cible sauf en ce qui concerne les classes. En fait, il n'y a pas de classe générée par le compilateur. Ce dernier combine le nom de l'instance de la classe avec le nom des différentes variables pour créer un nom de variable unique. Par exemple, imaginons une classe ABC avec deux variables x et y, et une instance A de cette classe. Les variables générées seront A_x et A_y.

En ce qui concerne les méthodes et les séquences, l'identificateur dans le programme cible LegOS est défini selon la forme « **graph_***id* », où *id* est l'identificateur qui a été attribué par le parser à la méthode ou la séquence. Cette dénomination fait référence au graphe de contrôle de flot et est conservée pour des raisons historiques.

5. Les types des variables

La figure 5.10 reprend les correspondances entre les types dSL et les types C. Par exemple, pour toute déclaration de variable dSL de type « **BOOL** », une variable de type « **unsigned char** » sera générée.

dSL	***C***
INT	short
BOOL	unsigned char
LONG	time_t

Figure 5.10 : Correspondance entre les types dSL et C.

6. Les variables

Il existe deux types de variables : les variables globales et les variables locales.

Nous avons vu que les variables globales d'un programme dSL sont distribuées statiquement sur un des sites du système (cf. *II.4.1. Localisation des variables*). Pour chacune de ces variables, sa déclaration est générée globalement sur le site sur lequel elle est distribuée. En ce qui concerne les variables globales tildées, leur déclaration se retrouve également sur les sites où la variable est utilisée de manière tildée. Remarquons qu'elles possèdent le même identificateur sur ces différents sites (cf. *10. La gestion des variables tildées*).

À ces déclarations globales viennent s'ajouter la déclaration des anciennes valeurs des conditions des événements (cf. *13. Les événements*). En effet, un événement se produit lorsque sa condition passe de vrai à faux. Il faut par conséquent conserver l'ancienne valeur de la condition de chaque événement. De plus, il sera généré les déclarations de variables booléennes d'état des séquences indiquant si la tâche est en attente ou non et dont la formulation et l'utilité seront expliquées plus loin (cf. *9.4. Les instructions d'attente*).

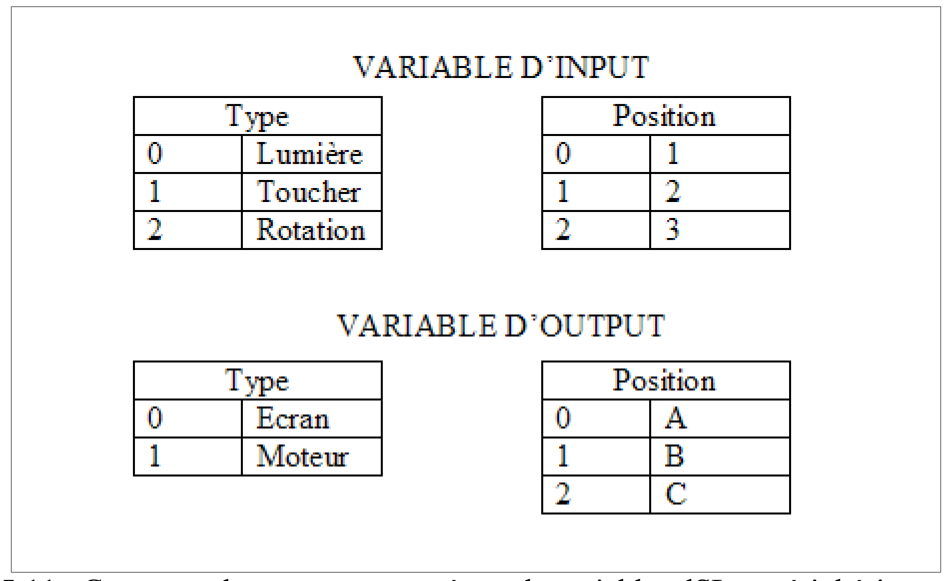
En résumé, pour un site donné, l'ensemble des variables globales définies est :
- les variables globales qui lui sont distribuées,
- les variables tildées utilisées sur le site,
- les anciennes conditions des événements,
- et les variables booléennes des états des séquences.

Les déclarations de ces variables sont générées dans le fichier .h des sites.

En ce qui concerne les variables locales (à une méthode ou à une séquence), leur déclaration sera située dans le scope de la fonction générée correspondante (cf. *11. Les méthodes* et *12. Les séquences*).

7. Les entrées/sorties

Avant de détailler la manière dont les dispositifs d'entrée et de sortie sont accédés, nous devons préciser la correspondance entre les paramètres dSL d'une variable globale d'entrée ou de sortie (cf. *II.8.7. Les sites et les entrées/sorties*) et les périphériques disponibles en legOS, à savoir les capteurs de toucher, de lumière et de rotation pour les entrées, l'écran LCD et les moteurs pour les sorties. La figure 5.11 reprend cette correspondance ainsi que la manière dont les emplacements sont spécifiés. Sur une brique Lego, il y a trois périphériques d'entrée, identifiés par les chiffres 1, 2 et 3, et trois périphériques de sortie, identifiés par les lettres A, B et C (cf. *III.1. Introduction aux Lego-Mindstorms*). Pour rappel, les paramètres dSL d'une variable d'entrée/sortie sont constitués de trois nombres : *card***.***rack***.***slot*. Le premier n'est pas utilisé, le second correspond à la position, le dernier au type du dispositif.



VARIABLE D'INPUT

Type	
0	Lumière
1	Toucher
2	Rotation

Position	
0	1
1	2
2	3

VARIABLE D'OUTPUT

Type	
0	Ecran
1	Moteur

Position	
0	A
1	B
2	C

Figure 5.11 : Correspondance entre paramètres de variables dSL et périphériques Lego.

Nous avons vu dans le paragraphe 2. *La tâche Input-Process-Output* que les fonctions *sampleInputs*() et *writeOutputs*() sont responsables de la gestion des entrées et des sorties du site. La fonction *sampleInputs*() consiste d'une part à relever les valeurs des dispositifs d'entrée (par l'intermédiaire de la fonction *sample*(*position*) qui renvoie la valeur du dispositif situé à l'emplacement *position*) et d'autre part, à les assigner à la variable globale correspondante. Remarquons qu'il s'agit d'une modification de la valeur d'une variable. Dès lors, si la variable est tildée, il est nécessaire de générer le code de transfert de valeur (cf. *10. La gestion des variables tildées*).

En ce qui concerne les variables de sortie, la gestion est un peu plus complexe. Pour rappel, deux types de dispositifs de sortie peuvent être utilisés : l'écran LCD et le moteur. Pour le premier, la fonction de mise à jour de la sortie est « **cputw(***valeur***)** », où *valeur* est la valeur de la variable associée à l'écran. Pour le second, la mise à jour se fait grâce à deux fonctions : « **motor_***x***_speed (***vitesse***)** » pour la vitesse et « **motor_***x***_dir(***direction***)** » pour la direction, où *x* est la position du moteur (a, b ou c). Pour être clair, imaginons un moteur en position A, sa mise à jour se fait grâce au fonction motor_a_speed(*vitesse*) et motor_a_dir(*direction*). La figure 5.12 reprend les différents paramètres de direction permis en LegOS. La direction est déterminée suivant le signe de la variable de sortie. Si celui-ci est positif, la direction est 1, s'il est négatif, la direction est 2, s'il est nul, la direction est 3. La vitesse sera la valeur absolue de la variable de sortie.

Paramètre	***Direction***
1	Marche avant
2	Marche arrière
3	Off (moteur en roue libre)

Figure 5.12 : Correspondance des directions des moteurs legOS.

8. Les opérateurs

L'opérateur d'assignation « **:=** » est traduit par l'opérateur C d'assignation « **=** ». L'opérateur « **MOD** » est traduit par l'opérateur modulo C « **%** ». Les opérateurs arithmétiques, relationnels et logiques sont traduits par le même mot-clé. En ce qui concerne l'opérateur de lancement, il nécessite un traitement particulier détaillé aux paragraphes *11. Les méthodes* et *12. Les séquences*.

9. Les instructions

9.1. Les instructions d'assignation

Une instruction d'assignation dSL

« *nom_variable* **:=** *expression* **;** »

est traduite par l'instruction C

« *nom_variable* **=** *expression_C* **;** »

où *expression_C* est la traduction C de *expression*.

Lorsqu'une assignation est effectuée sur une variable tildée, le compilateur doit générer le code de l'envoi de la nouvelle valeur de la variable aux sites sur lesquels cette variable est utilisée (cf. *10. La gestion des tildes*).

Lorsqu'une assignation est effectuée sur une variable apparaissant dans au moins une condition d'un événement du site, un appel à la fonction *handleWhen*() est généré (cf. *13. Les événements*).

9.2. Les instructions conditionnelles

Une instruction conditionnelle dSL utilisée en dehors d'une séquence

« **IF** *condition* **THEN** *liste_instructions_1* **ELSE** *liste_instructions_2* **END_IF** »

est traduite par l'instruction conditionnelle C

« **if (** *condition* **) {** *liste_instructions_1_C* **} else {** *liste_instructions_2_C* **}** »

où *liste_instructions_x_C* est la traduction C de *liste_instructions_x*.

De même, une instruction conditionnelle dSL en dehors d'une séquence

« **IF** *condition* **THEN** *liste_instructions_1* **END_IF ;** »

est traduite par l'instruction conditionnelle C

« **if (** *condition* **) {** *liste_instructions_1_C* **}** »

En ce qui concerne les instructions conditionnelles utilisées dans une séquence, la génération de code est un peu plus complexe car elle conserve la structure en blocs de base. Elle est expliquée dans le paragraphe *12. Les séquences.*

9.3. Les instructions d'itération

Une instruction d'itération dSL utilisée en dehors d'une séquence

« **WHILE** *condition* **DO** *liste_instructions* **END_WHILE ;** »

est traduite par l'instruction d'itération C

« **while (** *condition* **) {** *liste_instructions_C* **}** »

où *liste_instructions_C* est la traduction C de *liste_instructions*.

En ce qui concerne les instructions d'itération utilisées dans une séquence, la génération de code conserve la structure en blocs de base. Elle est expliquée dans le paragraphe *12. Les séquences*.

9.4. Les instructions d'attente

Une instruction d'attente n'a pas d'équivalent en C. Il faut donc générer du code C de comportement équivalent à celle d'une instruction d'attente.

Lors de l'exécution d'une instruction d'attente, la séquence est mise en attente jusqu'à ce que la condition soit vraie. Une séquence équivaut à une tâche. La condition est testée par la tâche de la séquence puisque elle peut contenir des variables locales. Si la condition est fausse, la tâche s'endort. À chaque assignation d'une variable de la condition, la tâche est réveillée pour qu'elle teste à nouveau la condition et se rendort si elle est fausse. Donc, à chaque fois qu'une variable apparaissant dans la condition d'une instruction d'attente d'une séquence est assignée, il faut
1. tester si la séquence est en attente,
2. et si oui, passer le contrôle à la séquence en attente.

1. Pour chaque séquence contenant au moins une instruction d'attente, une variable globale booléenne est définie. Elle indique si oui ou non la séquence est en train d'attendre. Ces variables sont appelées les variables d'état de séquences et sont générées selon la syntaxe « **bool __is_waiting***id_séquence* **;** » où *id_séquence* est l'identificateur de la séquence. Elles sont mises à vrai avant le test de la condition et mises à faux après le réveil de la séquence.

2. Puisque le contrôle est passé à une séquence, il doit être rendu à la tâche appelante. Lorsqu'une tâche effectue une modification sur une variable d'instruction d'attente et passe le contrôle à une séquence en attente, elle doit placer son identificateur de tâche sur le stack de contrôle (cf. *1.Les tâches*).

L'instruction d'attente dSL

« **WAIT** *condition* **;** »

est traduite par la suite d'instructions C reprise à la figure 5.13 où *id_sequence* est l'identificateur de la séquence.

« **__is_waiting***id_séquence* **= TRUE ;**
WHILE ! **(** *condition* **) { startwait(); }**
__is_waiting*id_séquence* **= FALSE ;** »

Figure 5.13 : Traduction d'une instruction d'attente.

Toute instruction générée d'assignation d'une variable apparaissant dans la condition d'une instruction d'attente est suivie par

« **if(__is_waiting***id_séquence***) wakeup(thread_***id_séquence***) ;** »

où *id_sequence* est l'identificateur de la séquence à réveiller.

Nous aurions pu envisager d'utiliser la notion d'événement LegOS et de générer l'expression booléenne de l'événement LegOS sur base de la condition d'attente de l'instruction dSL. C'est impossible puisque la condition peut contenir des variables locales.

10. La gestion des variables tildées

Une variable tildée est définie sur tous les sites sur lesquels elle est utilisée. À chaque assignation sur une variable tildée, le code du transfert de la nouvelle valeur doit être généré. La sémantique dSL impose un broadcast des messages de changement de valeur. Cependant, les fonctions de création et d'envoi de messages ainsi que le protocole de communication fiable sont définis tels que le message est adressé à un site de destination en particulier. En pratique, nous avons donc généré l'envoi du message pour chaque site où la variable est utilisée. Cet envoi se fait via l'instruction

« **SEND_ID_WARN (int** *site***, int** *id***, int** *val***) ;** »

Nous avons vu dans le paragraphe *3.3 Traitement des messages* que dans le cas de la réception d'un message de transfert de valeur d'une variable tildée, un appel à *handleWarn*(int *id*, int *val*) est exécuté dans *handleMsg*() où *id* est l'identificateur de la variable et *val* sa valeur. Le code de cette fonction est défini à la compilation. Il est généré dans le fichier .c du site sous la forme d'un switch sur l'identificateur de la variable. Pour chacune des variables tildées utilisées sur le site, un `case` est généré selon le format

« **case** *id_variable* **:** *nom_variable* **= val ;** »

où *id_variable* est l'identificateur de la variable, *nom_variable* son nom dans le programme dSL.

Lorsque le type de la variable est booléen, une conversion est effectuée en ajoutant « **(bool)** » avant « **val** ».

Remarquons que nous avons dû faire face à un petit problème d'efficacité. Imaginons une variable d'input tildée. À chaque fois que la boucle *Input-Process-Output* va être effectuée, un ou plusieurs messages seront envoyés en fonction du nombre de sites sur lesquels la variable est utilisée. Nous avons donc pris la précaution de déclarer localement à *sampleInputs*() une variable contenant l'ancienne valeur de la variable d'entrée tildée, c'est-à-dire la valeur de la variable avant d'avoir lu le périphérique d'entrée. Le message de transfert de valeur n'est envoyé uniquement s'il y a un changement de valeur de l'entrée.

11. Les méthodes

Pour toute méthode, une fonction sera générée. Si la méthode a des paramètres, la fonction prend en paramètre un pointeur de caractère `args`. Il contient les paramètres de la méthode, à savoir les valeurs de ceux-ci dans l'ordre de leur définition. Les identificateurs des paramètres ne sont pas nécessaires.

Donc, pour toute définition dSL d'une méthode distribuée sur un site,

« **METHOD** *classe* **::** *id_méthode* **(***paramètres***)** *liste_instructions* **END_METHOD** »

le compilateur va créer une fonction

« **void graph_***id***(***param***) {** *liste_instructions_C* **}** »

où *id* est l'identificateur de la méthode au sein du compilateur, *param* l'éventuelle liste de paramètres (ε si *paramètres* est ε, « **unsigned char * args** » sinon) et *liste_instructions_C* est la liste des instructions C de la méthode. Les instructions sont éventuellement précédées de la déclaration des variables locales ainsi que de la déclaration des paramètres et des instructions d'extraction des paramètres si *paramètres* n'est pas vide.

Une instruction d'appel synchrone d'une méthode sans paramètre est traduite par un appel direct à la fonction correspondante d'identificateur *id*

graph_*id***();**

Si la méthode prend des paramètres, les valeurs des paramètres sont stockés dans un vecteur de caractère transmis en paramètre à la méthode.

Dans le cas d'un appel asynchrone à une méthode, un message de demande d'exécution de méthode est envoyé. Une instruction d'appel asynchrone est traduite par

« **LAUNCH_ID_COLOR(int** *site***, int** *id***, int** *nombre_paramètres***,** *paramètres***);** »

où *id* est l'identificateur de la méthode, *site*, son site de distribution, *nombre_paramètres*, le nombre de paramètres à transmettre et *paramètres*, la liste des paramètres.

Lorsqu'un message d'appel d'une méthode est reçu, args est extrait du message par *handleMsg*() et transmis à la fonction *handleExec*(int *id*, unsigned char * args). Cette dernière est chargée d'appeler

la fonction graph_*id*(args) ou graph_*id*() suivant que la méthode possède ou non des paramètres. L'extraction des paramètres est effectuée dans le code de la méthode après la déclaration des paramètres et des variables locales.

Le code de *handleExec*(*id*,args) est défini à la compilation. Il est généré dans le fichier .c du site sous la forme d'un switch sur l'identificateur de la méthode. Pour chaque méthode d'un site, un `case` est généré selon le format

« **case** *id_méthode* **: graph_***id_méthode***(***param***); break;** »

où *id_méthode* est l'identificateur de la méthode et *param* est ε si la méthode n'a pas de paramètre, « **args** » sinon.

12. Les séquences

Le code d'une séquence est découpé par le compilateur en plusieurs blocs de base distribués sur plusieurs sites et reliés par des points de migration. Ces derniers correspondent à des messages de demande d'exécution d'une partie de séquence qui contiennent les valeurs des variables actives à mettre à jour dans le contexte local de la séquence. Au paragraphe *1. Les tâches*, nous avons vu que, pour des raisons de simplicité de gestion du contexte local, une séquence est traduite par une tâche continue. Pour toute séquence, une tâche est créée dès l'initialisation du système sur chaque site sur lequel au moins une de ses parties est distribuée. La fonction contenant le code de la tâche est identifiée de la même manière que les méthodes, c'est-à-dire par

« **void graph_***id***()** »

où *id* est l'identificateur de la séquence. Cette fonction ne prend jamais de paramètre. L'identificateur de la tâche créée est stocké dans une variable globale « **thread_***id* » dont l'utilité sera expliquée plus loin. Le code de la tâche contient 1. la déclaration des paramètres et des variables locales de la séquence et 2. le code de toutes les parties de la séquence distribuées sur le site. La tâche se met en attente de la réception d'un message de demande d'exécution d'une partie de la séquence. Une fois réveillée, elle exécute la partie de séquence demandée. Le pseudo code de la figure 5.14 exprime le comportement d'une tâche de séquence découpée en trois blocs de base dont le deuxième est distribué sur un autre site. *code_de_migration* correspond à la demande d'exécution de la deuxième partie de la séquence avec le transfert des éventuelles variables actives. *attente_message_reçu* correspond à l'attente de réception d'un message de demande d'exécution d'une partie de la séquence. *traitement_message* consiste en l'extraction des informations du message, à savoir la partie de séquence à exécuter et la valeur des paramètres ou des variables actives.

```
« void graph_id() {
    label_1 :
        code de la première partie de la séquence
    label_2 :
        code_de_migration
        goto dispatch;
    label_3 :
        code de la troisième partie de la séquence
    dispatch :
        attente_message_reçu
        traitement_message
        goto label(message)
  } »
```

Figure 5.14 : Pseudo code d'une tâche de séquence.

Le mécanisme goto n'étant pas disponibe sous legOS, nous l'avons mis en oeuvre par l'utilisation d'une boucle infinie contenant un switch sur l'identificateur de la partie de séquence à exécuter. La traduction en code C de la figure 5.14 est reprise à la figure 5.15.

```
« void graph_id() {
    Déclaration des paramètres
    Déclaration des variables locales
    int nextBasicBlock = HOP_WAIT_ID;
    while(true) {
      switch(nextBasicBlock) {
        case label_1 :
            code de la première partie de séquence
            HOP(id_site, id, label_2,  ... ) ;
            nextBasicBlock = HOP_WAIT_ID ;
            break;
        case label_3 :
            code de la troisième partie de séquence
            nextBasicBlock = HOP_WAIT_ID ;
            break;
        case HOP_WAIT_ID :
            startwait();
            mise à jour de nextBasicBlock
            extraction des paramètres
            break;
        case default :
            error(ERR_UNKNOWN_HOP_ID) ;
            halt();
      }
    }
  } »
```

Figure 5.15 : Code d'une tâche de séquence.

Deux `case` supplémentaires sont ajoutés par séquence : « **case default** » en cas d'erreur, c'est-à-dire en cas d'identificateur de bloc erroné, et « **case HOP_WAIT_ID** ». Ce dernier correspond à dispatch, c'est-à-dire à l'état d'attente d'une séquence. Il contient 1. l'appel de *startwait*() qui met la

tâche courante en attente de l'événement de reprise du contrôle ; 2. l'identification de la partie de séquence à exécuter ; 3. les instructions d'extraction des variables actives ou des paramètres lorsqu'il s'agit de la première partie de la séquence. L'identificateur du bloc à exécuter et les paramètres sont transmis de *handleMsg*() à la tâche par l'intermédiaire de deux variables globales, respectivement **dsl_vm_id** et **dsl_vm_args**.

Après le traitement du message et la mise à jour de ces variables, *handleMsg*() réveille la tâche de séquence. Pour réveiller une tâche, la fonction *handleMsg*() a besoin de connaître son identificateur. L'identificateur est attribué lorsque la tâche est créée en début d'exécution. Pour cela, nous avons créé une variable globale « **pid_t thread_***id* » par séquence qui stocke l'identificateur de la tâche de séquence d'identificateur *id*. La fonction *getID*() est générée à la compilation telle que, sur base de l'identificateur de la séquence, elle renvoie l'identificateur de la tâche correspondante. La figure 5.16 présente la structure de la fonction *getID*().

```
pid_t getID(int id) {
  switch(id) {
     ...
     case  id_x  :  return thread_id_x;
     ...
     case default : error(ERR_UNKNOWN_ID);
}
```

Figure 5.16 : Structure de la fonction getID()

Lorsqu'un message de saut est reçu, la tâche correspondante est réveillée. Le `current_thread_id` est assigné à celui de la séquence et l'identificateur du `case` à exécuter (extrait du message) est placé dans la variable globale `dsl_vm_id`. La séquence se réveille dans le case HOP_WAIT_ID et consulte `dsl_vm_id` pour savoir quel est l'identificateur du bloc à exécuter ainsi que `dsl_vm_args` pour récupérer les variables transmises en paramètre.

Le lancement d'une séquence dans le code dSL est traduit par l'instruction

« **HOP(***site*, *id_séquence*, *id_partie*, *paramètres***)** »

où *id_partie* est l'identificateur de la première partie de la séquence identifiée par *id_séquence*, *site* est le site de distribution de cette partie et *paramètres*, sont les paramètres éventuels de la séquence. Lorsqu'il s'agit de traduire un point de migration, c'est-à-dire lorsque l'on se trouve à l'intérieur d'une séquence et qu'il faut ordonner l'exécution de la partie de séquence suivante située sur un autre site, le compilateur génère également un appel à la fonction HOP(*site*, *id_séquence*, *id_partie*, *paramètres*) où *paramètres* correspond aux variables actives à transmettre à la partie de séquence suivante ainsi que la mise à jour du prochain `case` à exécuter, à savoir le HOP_WAIT_ID puisque la tâche doit être mise en attente.

Lorsque la séquence contient des instructions conditionnelles ou d'itération, la structure en blocs de base est conservée. La mise à jour du prochain `case` à exécuter se fait donc en fonction du test de la condition.

Une instruction conditionnelle dSL dans une séquence

« **IF** *condition* **THEN** *liste_instructions_1* **ELSE** *liste_instructions_2* **END_IF** »

est traduite par les intructions reprises dans la figure 5.17 où *liste_instructions_x_C* est la traduction C de *liste_instructions_x* et *label_y* contient le code des instructions suivant l'instruction conditionnelle dSL.

« **case** *label_x* :
  ...
  if (*condition***) nextBasicBlock** = *label_1* **;**
  else nextBasicBlock = *label_2* **;**
  break;
case *label_1* **:**
  liste_instructions_1_C
  nextBasicBlock = *label_y* **;**
  break;
case *label_2* **:**
  liste_instructions_2_C
  nextBasicBlock = *label_y* **;**
  break;
case *label_y* **:**
  ... »

Figure 5.17 : Traduction d'un instruction conditionnelle dans une séquence.

Une instruction d'itération dSL dans une séquence

« **WHILE** *condition* **DO** *liste_instructions* **END_WHILE ;** »

est traduite par la suite d'instructions reprises dans la figure 5.18 où *liste_instructions_C* est la traduction C de *liste_instructions* et *label_y* contient le code des instructions suivant l'instruction d'itération dSL.

« **case** *label_x* :
  if (*condition***) nextBasicBlock** = *label_z* **;**
  else nextBasicBlock = *label_y* **;**
  break;
case *label_z* **:**
  liste_instructions_C
  nextBasicBlock = *label_x* **;**
  break;
case *label_y* **:**
  ... »

Figure 5.18 : Traduction d'une instruction d'itération dans une séquence.

13. Les événements

Soit l'événement suivant

« **WHEN** *condition* **THEN** *liste_instructions* **END_WHEN** »

liste_instructions doit être exécuté chaque fois que *condition* passe de faux à vrai. Une variable

globale booléenne est créée pour chaque événement d'un site et contient l'ancienne valeur de la condition de l'événement (cf. *6. Les variables*).

Les instructions de traitement d'un événement sont isolées dans une fonction « **void graph_*id*()** ». Cette manière de procéder est conservée pour des raisons historiques. Elle provient du fait que le parser crée un bloc de base d'identificateur *id* qui contient le traitement de l'événement. Le code de l'inspection de la condition d'un événement est repris dans la figure 5.19 où *id_événement* est l'identificateur de l'événement et *id_traitement* est l'identificateur du bloc de base des instructions de l'événement.

```
« IF ( NOT old_condition_id_événement AND condition ) {
      old_condition_id_événement = condition ;
      graph_id();
   } else {      old_condition_id_événement = condition ;    } »
```

Figure 5.19 : Code d'inspection d'une codition.

La condition d'un événement doit être inspectée chaque fois qu'une des variables apparaissant dans l'expression de sa condition est modifiée. Lorsqu'une variable apparaissant dans le code d'au moins une condition d'événement est assignée,

1. soit on inspecte uniquement les conditions des événements dans lesquelles la variable apparaît. Lors de la compilation, il faut conserver, pour toute variable globale, les identificateurs des blocs de base d'inspection des conditions d'événement dans lesquels la variable apparaît. Cette solution est donc plus complexe, génère plus de code mais est plus efficace en temps d'exécution.

2. soit on inspecte systématiquement les conditions de tous les événements du site. Ce qui simplifie la gestion des événements, génère moins de code mais est moins efficace puisque certaines des conditions inspectées pourront ne pas avoir changé de valeur.

La deuxième option a été retenue vu le peu de place mémoire disponible sur les Lego-Mindstorms. Chaque assignation sur une variable d'une condition d'évenement est suivie d'un appel à *handleWhens*(), la fonction contenant le code de l'inspection de toutes les conditions des événements du site.

La gestion des événements implique donc la génération de code par le compilateur
1. d'une fonction de traitement pour chaque événement du site,
2. et de la fonction *handleWhens*().

À chaque fois qu'une variable sur laquelle porte au moins un événement est assignée, *handleWhens*() est exécutée (cf. *6. Les variables*). De plus, elle est exécutée après chaque cycle de sampling des entrées (cf. *4. La machine virtuelle*).

14. Exemples de génération de code

Ce paragraphe présente une série d'exemples de code généré pour l'ensemble des constructions dSL. La structure de celui-ci est calquée sur le paragraphe *II.9 Exemples*. La grande majorité des exemples sont d'ailleurs tirés de ce paragraphe. Cela dit, quelques exemples supplémentaires plus représentatifs ont été ajoutés.

14.1. Les tâches

Pour illustrer la création de tâches, nous allons présenter un exemple de fonction d'initialisation générée par le compilateur repris dans la figure 5.20. Elle correspond à un site sur lequel ont été distribuées deux séquences d'identificateur respectif 2 et 3. Trois tâches sont donc créées. threadStart correspond à la tâche *Input-Process-Output* à laquelle est assignée une priorité de 8, thread_2 et thread_3 sont les identificateurs des tâches des séquences de priorité 9. graph_2() et graph_3() contiennent les instructions des séquences. Nous pouvons remarquer que les tâches de séquence sont initialisées et mises en attente d'un message avant que la tâche *Input-Process-Output* ne reçoive le contrôle de l'OS.

```
int main() {
   init();
   init_AB_protocol(portHandler);
   threadStart=execi(&start, 0, 0, PRIO_NORMAL-2, DEFAULT_STACK_SIZE/2);
   thread_2=execi(&graph_2,0,0,PRIO_NORMAL-1,DEFAULT_STACK_SIZE/2);
   thread_3=execi(&graph_3,0,0,PRIO_NORMAL-1,DEFAULT_STACK_SIZE/2);
   return 0;
}
```

Figure 5.20 : Exemple de fonction d'initialisation générée.

14.2. Les variables

La figure 5.21 reprend un exemple de déclaration de variables globales distribuées sur deux sites. Elles sont générées à partir du code dSL de la figure 2.13. Les variables distribuées sur le site 1 auront leur déclaration dans le fichier vm_1.h généré par le compilateur, idem pour le site 2.

```
vm_1.h :

   short entree1;
   unsigned char sortie1;
   time_t interne1;

vm_2.h :

   short entree2;
   unsigned char sortie2;
```

Figure 5.21 : Exemples de déclaration de variables globales.

La figure 5.22 reprend un exemple de déclaration de variables de classe correspondant à la figure 2.12.

```
int v1_vitesse;
bool v1_marche;
int v2_vitesse;
bool v2_marche;
```

Figure 5.22 : Exemple de déclaration de variables globales générées.

14.3. Les entrées/sorties

La figure 5.23 reprend un exemple de déclaration de variables dSL correspondant à tous les types de dispositifs disponibles. La figure 5.24 présente le contenu des fonctions *sampleInputs*() et *writeOutputs*() générées.

```
INPUT lumière : 0.0.0 ; (*position 1*)
INPUT toucher : 0.1.0 ; (*position 2*)
INPUT rotation: 0.2.0 ; (*position 3*)
OUTPUT moteur : 1.1.1 ; (*position B*)
OUTPUT lcd : 1.0.0 ;
```

Figure 5.23 : Exemple de déclaration d'entrées/sorties dSL.

```
void sampleInputs() {
   lumière = sample(1);
   toucher = sample(2);
   rotation = sample(3);
}

void writeOutputs() {
   cputw(lcd);
   if(moteur<0) motor_b_dir(2);
   else  if(moteur>0) motor_b_dir(1);
   else  motor_b_dir(3);
   motor_b_speed(ABS(moteur));
}
```

Figure 5.24 : Exemple de mise à jour des dispositifs.

14.4. Les instructions d'assignation

La figure 5.25 reprend des exemples d'instructions d'assignation correspondant à l'exemple de la figure 2.18.

```
entier = 0 ;
entier = ( 1 + 2 - 3 ) % ( 10 ) ;
entier = entier + 1 ;
entier = LONG_TO_INT(long) + BOOL_TO_INT(booleen) ;

booleen = TRUE ;
booleen = ( LONG_TO_BOOL(long) OR INT_TO_BOOL(entier) );
```

Figure 5.25 : Exemples d'instructions d'assignation générées.

14.5. Les instructions conditionnelles

La génération de code pour les instructions conditionnelles dSL de la figure 2.20 est reprise à la figure 5.26. Ces instructions sont traitées comme situées en dehors d'une séquence.

```
if ( entier>10 ) {
     booleen = TRUE;
     ...
}

if ( entier>10 ) {
     booleen = TRUE;
} else {
     booleen = FALSE;
}

if ( entier>10 ) {
   if ( booleen==FALSE ) {
         booleen = TRUE;
   }
} else {
   if ( booleen==TRUE ) {
         booleen=FALSE;
   }
}
```

Figure 5.26 : Exemple d'instructions conditionnelles générées.

14.6. Les instructions d'itération

Le code généré de l'instruction d'itération dSL de la figure 2.21 est repris à la figure 5.27. Cette instruction est considérée comme définie en dehors d'une séquence.

```
while ( booleen ) {
     entier = entier+1;
     if ( entier>10 ) {
          booleen = TRUE;
     }
}
```

Figure 5.27 : Exemple d'instructions conditionnelles.

14.7. Les méthodes et les instructions de lancement

Nous allons reprendre ici les deux méthodes de la figure 2.23 afin d'illustrer la génération de code pour les méthodes. Les identificateurs des méthodes sont par exemple 4 pour accélérer(ajout) et 5 pour freiner(). Deux fonctions vont donc être créées, l'une avec paramètre, l'autre sans. La figure 5.28 présente le code généré pour ces deux fonctions. La figure 5.29 montre le code généré pour la fonction *handleExec*(args). La fonction READ_INT_FROM_CHAR(int, unsigned char*) est une macro définie dans dsl_vm.h.

```
void graph_4(unsigned char * args) { /*accélérer*/
   int ajout;
   READ_INT_FROM_CHAR(ajout,args);
   v1_vitesse = v1_vitesse + ajout;
}

void graph_5() { /*freiner*/
     v1_vitesse = 0 ;
}
```

Figure 5.28 : Exemple de déclarations de méthodes générées.

```
void handleExec(int id, unsigned char* args) {
     switch(id) {
          case 4 : if(args) graph_4(args);
                   break;
          case 5 : graph_5();
                   break;
          case default : error(ERR_UNKNOWN_ID);
                   halt();
      }
}
```

Figure 5.29 : Exemple de fonction handleExec().

La figure 5.30 reprend la génération de code des appels dSL aux méthodes présentées ci-dessus de la figure 2.24. Les méthodes sont distribuées sur le site d'identificateur 1. La première instruction appelle freiner() asynchroniquement, la deuxième et la quatrième appelle accélérer() asynchroniquement avec comme paramètre 2 et `plus`. La troisième est un appel synchrone à avancer (). `dsl_vm_arg` est un vecteur de caractère défini globalement dans dsl_vm.h. Il n'est utilisé que pour les appels synchrones. La fonction ADD_TO_CHAR(unsigned char*, int) est une macro définie dans dsl_vm.h.

```
LAUNCH_ID_COLOR(1,5,0);

LAUNCH_ID_COLOR(1,4,1,2);

ADD_TO_CHAR(dsl_vm_arg,plus);
graph_5(dsl_vm_arg);

LAUNCH_ID_COLOR(1,4,1,plus);
```

Figure 5.30 : Exemples d'appels de méthodes générés.

14.8. La gestion des variables tildées

La figure 5.31 reprend un exemple de génération de code d'une fonction *handleWarn*(id,val) pour un site utilisant deux variables tildées x et y, respectivement d'identificateur 25 et 26.

```
void handleWarn(int id, int val) {
    switch(id) {
        case 25 : x = val;
                  break;
        case 26 : y = val;
                  break;
        case default : error(ERR_UNKNOWN_WARN_ID);
                       halt();
    }
}
```

Figure 5.31 : Exemple de génération de code pour une fonction *handleWarn*().

14.9. Les séquences

Un exemple de séquence distribuée sur trois sites est présenté dans la figure 2.29. Les figures 5.32, 5.33 et 5.34 reprennent le code généré correspondant à cette séquence, respectivement pour le site 1, 2 et 3. Remarquons la génération de code pour une instruction conditionnelle dans une séquence.

Une analyse de variables actives permet de se rendre compte que la variable `l` d'identificateur 23 doit être transmise lors de la demande d'exécution du bloc de base d'identificateur 14 distribué sur le site 1. Le code de la demande d'exécution se trouve dans le case 13 de la figure 5.33. Le code d'extraction des paramètres a donc été généré pour le site 1.

```
void graph_6() { /*distri*/
    int l;
    int nbVal;
    int nextBasicBlock = HOP_WAIT_ID;
    while (true) {
        switch(nextBasicBlock) {
            case HOP_WAIT_ID:
                startwait();
                nextBasicBlock=dsl_vm_id;
                nbVal=READ_INT_FROM_CHAR(dsl_vm_args);
                while(nbVal--) {
                    switch(READ_INT_FROM_CHAR(dsl_vm_args)) {
                        case 23 :
                            l=READ_INT_FROM_CHAR(dsl_vm_args);
                            break;
                        case default :
                            error(ERR_UNKNOWN_PARAM_ID);
                            halt();
                    }
                }
                break;
            case 10 :
                if(a>0) nextBasicBlock=11;
                else nextBasicBlock=12;
                break;
            case 11 :
                b=1;
                HOP(2,6,13,0);
                nextBasicBlock = HOP_WAIT_ID;
                break;
            case 12 :
                b=0;
                HOP(2,6,13,0);
                nextBasicBlock = HOP_WAIT_ID;
                break;
            case 14 :
                b=a+b+l;
                HOP(3,6,15,0);
                nextBasicBlock = HOP_WAIT_ID;
                break;
            case 16 :
                g=a;
                b=l;
                nextBasicBlock = HOP_WAIT_ID;
                break;
         }
    }
}
```

Figure 5.32 : Exemple de code généré pour une séquence distribuée – 1ère partie.

```
void graph_6() { /*distri*/
    int l;
    int nextBasicBlock = HOP_WAIT_ID;
    while (true) {
        switch(nextBasicBlock) {
            case HOP_WAIT_ID:
                startwait();
                nextBasicBlock=dsl_vm_id;
                break;
            case default :
                error(ERR_UNKNOWN_HOP_ID);
                halt();
            case 13 :
                g=c;
                c=d;
                d=l;
                HOP(1,6,14,1,23,l);
                nextBasicBlock = HOP_WAIT_ID;
                break;
        }
    }
}
```

Figure 5.33 : Exemple de code généré pour une séquence distribuée – 2ème partie.

```
void graph_6() { /*distri*/
    int l;
    int nextBasicBlock = HOP_WAIT_ID;
    while (true) {
        switch(nextBasicBlock) {
            case HOP_WAIT_ID:
                startwait();
                nextBasicBlock=dsl_vm_id;
                break;
            case default :
                error(ERR_UNKNOWN_HOP_ID);
                halt();
            case 15 :
                e=f;
                HOP(1,6,16,0);
                nextBasicBlock = HOP_WAIT_ID;
                break;
        }
    }
}
```

Figure 5.34 : Exemple de code généré pour une séquence distribuée – 3ème partie.

La séquence présentée çi-dessus ne prend pas de paramètre. La séquence de la figure 2.30 prend deux paramètres `x` et `y`. Le code généré pour cette séquence est repris dans la figure 5.35. Nous profiterons aussi de l'occasion pour illustrer la génération de code associée à une instruction d'attente dSL. Supposons que le compilateur doit générer du code pour l'instruction « `i:=TRUE;` ». Cette assignation nécessite de tester l'état de la séquence 6. Si celle-ci est en train d'attendre, il faut lui passer le contrôle.

```
void graph_7() { /*distri*/
    int x;
    bool y;
    int nbVal;
    int nextBasicBlock = HOP_WAIT_ID;
    while (true) {
        switch(nextBasicBlock) {
            case HOP_WAIT_ID:
                startwait();
                nextBasicBlock=dsl_vm_id;
                nbVal=READ_INT_FROM_CHAR(dsl_vm_args);
                while(nbVal--) {
                    switch(READ_INT_FROM_CHAR(dsl_vm_args)) {
                        case 24 :
                            x=READ_INT_FROM_CHAR(dsl_vm_args);
                            break;
                        case 25 :
                            y=READ_BOOL_FROM_CHAR(dsl_vm_args);
                            break;
                        case default :
                            error(ERR_UNKNOWN_PARAM_ID);
                            halt();
                    }
                }
                break;
            case default :
                error(ERR_UNKNOWN_HOP_ID);
                halt();
            case 18 :
                __is_waiting6 = TRUE;
                while(!(i OR y)) {
                    startwait();
                }
                __is_waiting6 = FALSE;
                h=x;
                nextBasicBlock = HOP_WAIT_ID;
                break;
        }
    }
}


...
i = TRUE;
if(__is_waiting6) wakeup(thread_6);
...
```

Figure 5.35 : Exemple de code généré pour une séquence à deux paramètres et avec une instruction d'attente.

14.10. Les événements

La figure 5.36 présente l'extrait du fichier vm_1.c correspondant à la série d'événements de la figure 2.31. La figure 5.37 reprend le code de vm_2.c.

```
void graph_10() { /*bEGALa*/
    int nextBasicBlock = HOP_WAIT_ID;
    while (true) {
        switch(nextBasicBlock) {
            case HOP_WAIT_ID:
                startwait();
                nextBasicBlock=dsl_vm_id;
                break;
            case default :
                error(ERR_UNKNOWN_HOP_ID);
                halt();
            case 19 :
                b = a;
                nextBasicBlock = HOP_WAIT_ID;
                break;
         }
    }
}

void graph_11() {
    b=1;
}

void graph_13() {
    HOP(1,9,19,0);
}

void graph_15() {
    LAUNCH_ID_COLOR(2,20,0);
}

void handleWhens() {
    BOOL cond_result;
    cond_result = (a>0);
    if (!old_condition_3 && cond_result) {
        old_condition_3 = cond_result;
        graph_11();
    } else { old_condition_3 = cond_result;  }
    cond_result = (a<0);
    if (!old_condition_3 && cond_result) {
        old_condition_3 = cond_result;
        graph_13();
    } else { old_condition_3 = cond_result;  }
    cond_result = (a==0);
    if (!old_condition_3 && cond_result) {
        old_condition_3 = cond_result;
        graph_15();
    } else { old_condition_3 = cond_result;  }
}
```

Figure 5.36 : Exemple de génération de code pour des événements – 1ère partie.

```
void graph_20() { /*cEGALd*/
   cd_c = cd_d ;
}

void graph_21() {
    graph_20();
}

void graph_22() {
    graph_20();
}

void handleWhens() {
    BOOL cond_result;
    cond_result = (c==0);
    if (!old_condition_3 && cond_result) {
        old_condition_3 = cond_result;
        graph_21();
    } else { old_condition_3 = cond_result;  }
    cond_result = (a==0);
    if (!old_condition_3 && cond_result) {
        old_condition_3 = cond_result;
        graph_22();
    } else { old_condition_3 = cond_result;  }
}
```

Figure 5.37 : Exemple de génération de code pour des événements - 2ème partie.

15. Le protocole de communication

La mise en œuvre de systèmes distribués nécessite un protocole fiable de communication. Il est impensable en effet qu'un message de saut dans une séquence soit perdu. N'oublions pas non plus que les systèmes implémentés sont critiques. Dans le cas d'une usine de traitement de produits chimiques, il est aisé d'imaginer que la perte d'un message de demande d'exécution d'une procédure d'évacuation est inconcevable.

15.1. Fonctionnement

A notre connaissance, aucun protocole fiable n'a été implémenté pour les Lego-Mindstorms. Nous avons donc pris la décision d'en implémenter un par nous-même. Ce protocole est basé sur le protocole d'*alternating bit* (appelé par la suite protocole ab). Il est adéquat par rapport à la typologie du réseau. En effet, le réseau infrarouge est sans mémoire. Les messages envoyés ne sont pas retardés. Ils se propagent sans encombre dans le réseau.

Le protocole est fiable à 100%. En effet, un message ne sera jamais considéré comme transmis alors qu'il ne l'a pas été. Si maintenant, un site n'est pas en état de recevoir un message (liaison infrarouge impossible ou crash du site), le message va être continuellement renvoyé.

Le protocole ab est très simple. Il s'agit en fait d'un protocole à fenêtre glissante de taille 1. Pour un émetteur et un récepteur donnés, il y a constamment un seul message en cours d'envoi. Le numéro identifiant le message alterne entre 0 et 1.

Soit une station d'envoi et une station de réception, chacune maintient un bit, à savoir respectivement un bit d'envoi et de réception, qui permet d'identifier la véracité du message. Ces

bits sont initialisés à la même valeur de part et d'autre (typiquement 0). Le réseau est supposé vide à l'initialisation. Lorsque la station d'envoi prépare son message à envoyer, elle y insère le bit d'envoi en question. La station réceptrice vérifie le bit contenu dans l'en-tête. Si celui-ci correspond à son bit de réception, le message est débarrassé de son en-tête et transmis à l'utilisateur. Le bit sur la station de réception est inversé. Dès réception d'un message, un acquittement contenant le bit du message est envoyé, même si le bit du message reçu ne correspond pas. Lorsque la station émettrice reçoit l'acquittement, elle vérifie sa véracité en comparant le bit avec celui du message envoyé et inverse son bit d'envoi si le message a bien été acquitté. La station émettrice peut alors passer à l'envoi d'un nouveau message. Lorsqu'aucun acquittement n'est reçu après un certain temps ou lorsque la station émettrice reçoit un acquittement avec un bit incorrect, le message en cours est renvoyé. Dans la figure 5.38, sont représentés différents cas qui peuvent se poser dans l'usage d'un protocole ab. [*i*]MSG(*j*) correspond au message *j* envoyé avec avec un bit = *i*. TEMPO signifie que le temps d'attente d'un acquittement est écoulé et qu'il faut réenvoyer le message.

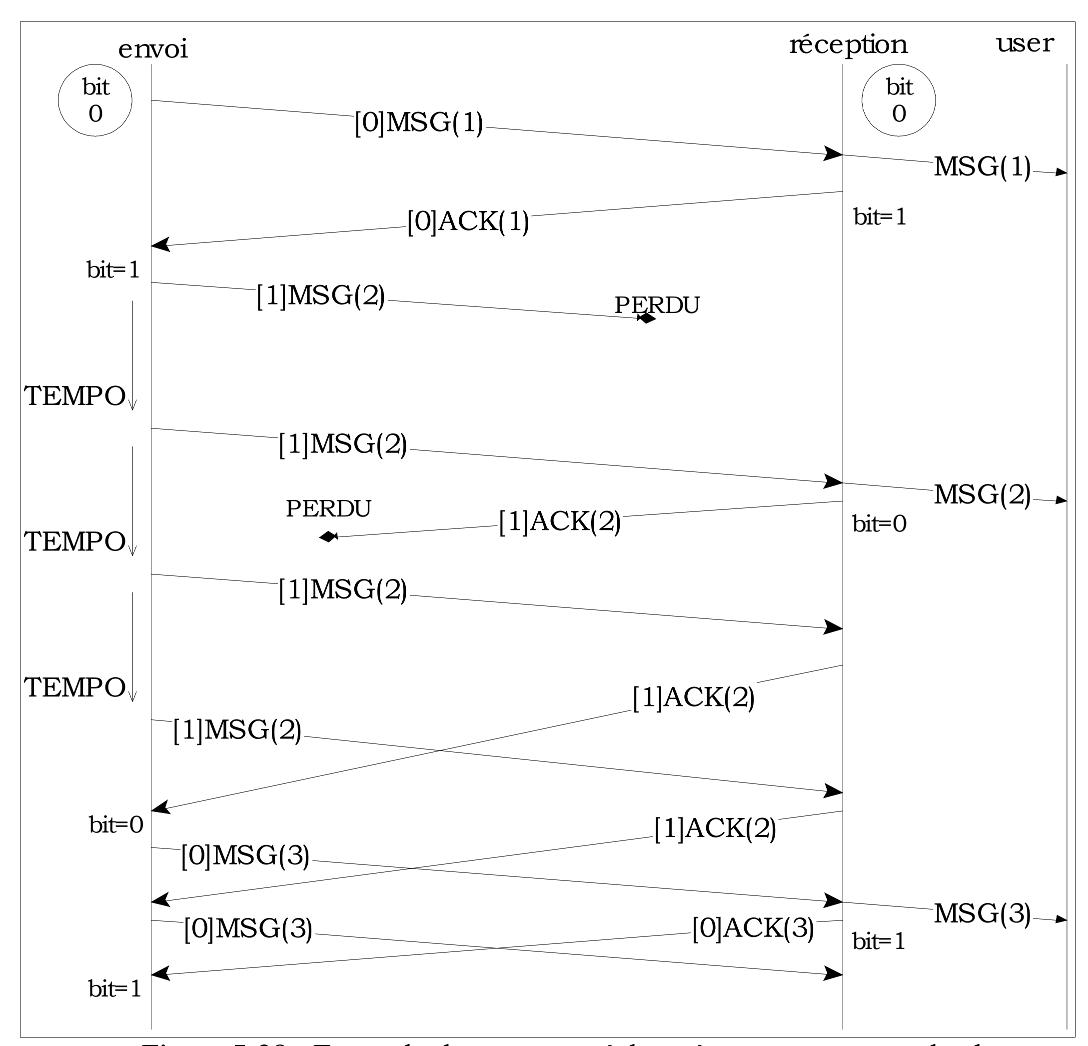


Figure 5.38 : Exemple de messages échangés avec un protocole ab.

Une station sera amené à envoyer et à recevoir des messages. Il faut donc voir les communications entre deux stations comme divisées en deux canaux. Un canal d'envoi et un canal de réception. De même, il faut que chaque station retienne un bit d'envoi et de réception pour chacune des stations avec laquelle elle communique.

La figure 5.39 reprend le pseudo code du protocole où [message,b] désigne la réception d'un message contenant le bit b ; (message,b) correspond à l'envoi de message avec le bit b ; maj

TEMPO signifie que le temporisateur est remis à zéro et [tempo] indique que le temps d'attente d'un acquittement est dépassé. De chaque côté, le bit est mis à zéro dès l'intialisation. Lorsqu'un message est reçu par une station, celle-ci compare le bit de l'en-tête du message avec le bit de réception correspondant à la source du message. Si ceux-ci coïncident, le message est transmis au programme utilisateur. Quoiqu'il en soit, il faut acquitter le message. Du côté de la station émettrice, lorsque le programme utilisateur lui donne un message à envoyer, elle ajoute l'en-tête du message contenant le bit d'envoi correspondant à la station réceptrice et envoie le message. Si un acquittement est reçu, le bit fourni par celui-ci est vérifié. S'il coïncide avec le bit d'envoi, le message a bien été transmis. Sinon, il faut renvoyer le dernier message envoyé.

```
RECEPTION :
– bit = 0 ;
– [message,b] :  b = bit : (ack,b), bit = 1 – bit ;
                 b ≠ bit : (ack,b)
ENVOI :
– bit = 0 ;
– envoi de message : (message, bit) + maj TEMPO
– [ack,b] :  b = bit : bit = 1 – bit ;
             b ≠ bit : (message, bit) + maj TEMPO
– [tempo] : (message, bit)
```

Figure 5.39 : Pseudo code du protocole ab.

15.2. Implémentation

Les messages sont tous broadcastés. Le protocole de communication est donc chargé de vérifier la destination du message et de transmettre à la fonction de traitement des messages reçus uniquement les messages qui lui sont destinés.

L'objectif de cette démarche est d'obtenir un protocole fiable tout en veillant à ce que l'implémentation soit réutilisable. En LegOS, une fonction (appelée *portHandler*) est assignée à la tâche de réception des messages. Dès qu'un message est reçu sur le port infrarouge, le système d'exploitation lance l'exécution du portHandler qui doit être défini par l'utilisateur et qui a pour rôle de traiter les messages. Imaginons maintenant qu'un utilisateur désire utiliser un protocole de communication fiable. La démarche naturelle est de remplacer le portHandler de l'utilisateur courant par un portHandler assurant la fiabilité des messages et transmettant les messages corrects au portHandler de l'utilisateur pour un traitement adéquat. La figure 5.40 reprend les hiérarchies de couches sans ou avec protocole ab.

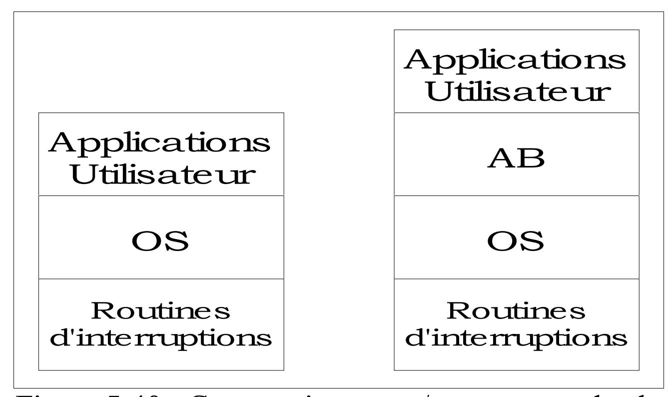


Figure 5.40 : Comparaison sans/avec protocole ab.

Sur un site donné, pour chaque site avec lequel il y aura échange de message (envoi ou réception), il faut conserver :

1. l'identificateur du site,
2. la liste des messages à envoyer (contenant le message en cours d'envoi),
3. le bit d'envoi,
4. et le bit de réception.

Ces éléments sont stockés dans une structure sous forme d'une liste de site. Pour chaque site, l'identificateur du site (ID_SITE), les bits d'envoi (S_BIT) et de réception (R_BIT) ainsi que la liste des messages à envoyer (LIST) sont stockés. Cette dernière contient les suites des caractères composant les messages (MSG) et leur longueur (LENGTH). Une représentation de cette structure est détaillée à la figure 5.41.

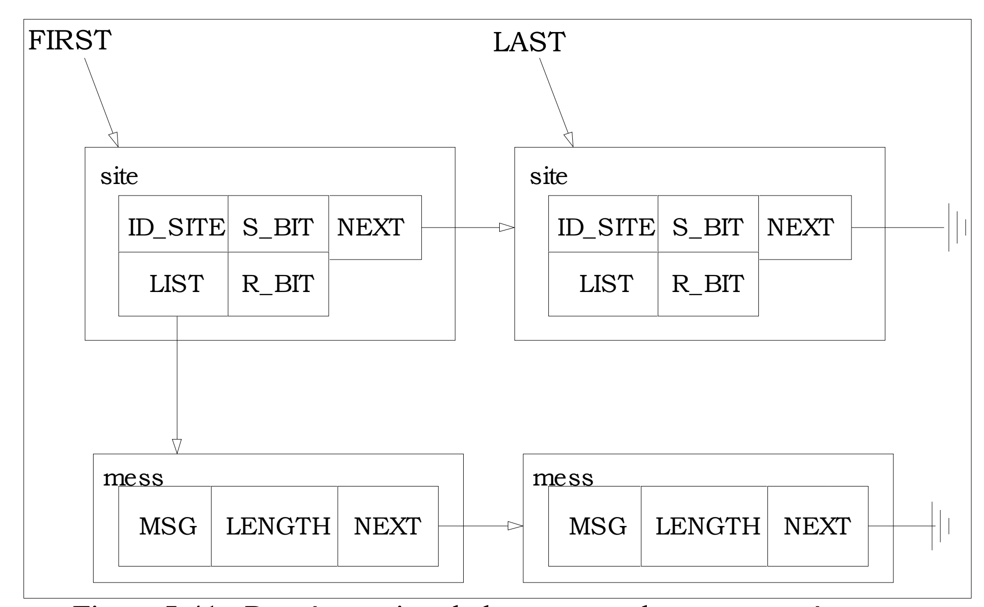


Figure 5.41 : Représentation de la structure des messages à envoyer.

Dans notre implémentation, nous avons prévu deux types d'envoi : synchrone et asynchrone.

L'envoi synchrone d'un message se fait via la fonction

« ab_swrite(int *site*, **unsigned char*** *message*, **int** *length***)** »

où *site* est la destination du message, *message*, le message et *length*, sa longueur.

L'envoi asynchrone d'un message se fait via la fonction

« ab_awrite(int *site*, **unsigned char*** *message*, **int** *length***)** »

L'envoi synchrone assure que le message est transmis à la fin de l'exécution de la fonction. Les deux fonctions 1. créent l'en-tête du message et forme le message à envoyer ; 2. cherchent la liste de messages du site de destination en la créant si besoin est et 3. ajoutent le message avec en-tête en fin de liste des messages à envoyer. L'ajout en fin de liste préserve l'ordre des messages. Le format général des messages ab est repris à la figure 5.42. En pratique, l'en-tête est constituée de trois bytes. Un byte concernant l'identification du message (contenant le bit d'envoi (`B`) et le bit d'acquittement (`A`)), et deux bytes identifiant la source et la destination du message.



Figure 5.42 : Format des messages ab.

Le chemin parcouru par un message envoyé est repris à la figure 5.43. Pour cela, différentes tâches vont s'exécuter en parallèle : 1. les applications utilisateur 2. le système d'exploitation (avec notamment les routines d'interruption) et 3. l'envoi des messages. Les structures de contrôle détaillées plus haut sont donc des ressources partagées entre ces tâches. Il faut donc mettre en œuvre une section critique. Le mécanisme de sémaphores assure l'exclusion mutuelle entre des tâches. Le sémaphore `sem_ab_write` est initialisé avec une valeur de 1. Au début de la fonction d'envoi, un `sem_wait(sem_ab_write)` permet d'attendre que l'exécution des autres instances de `ab_write` soient terminées. Un `sem_post(sem_ab_write)` en fin de fonction permet de libérer le sémaphore pour qu'une autre instance de la fonction puisse s'exécuter.

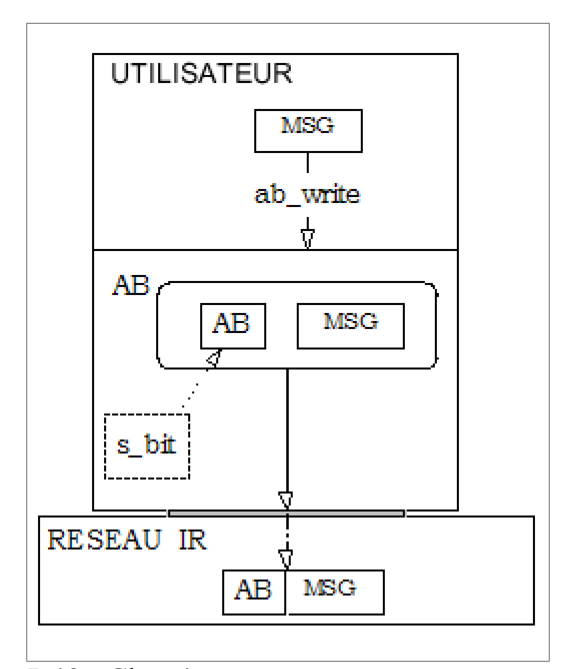


Figure 5.43 : Chemin parcouru par un message envoyé.

Lors de la réception d'un message sur le port infrarouge, le système d'exploitation donne le contrôle au portHandler et lui transmet le message reçu. Les messages envoyés sont tous broadcastés. La première mission du portHandler est donc de trier les messages pour ne conserver que ceux destinés au site sur lequel il se trouve. Le portHandler constitue une routine d'interruption, il ne peut donc être mis en attente. Cela implique qu'il ne peut pas envoyer lui-même l'acquittement d'un message reçu. Les messages reçus sont donc stockés dans une liste en attendant d'être traité. La structure de la lise des messages reçus est reprise à la figure 5.44. Puisqu'il accède à une structure de données, il faut assurer l'exclusion mutuelle par l'usage de sémaphore. Mais puisque le portHandler ne peut être mis en attente, c'est `sem_trywait(sem_ab_write)` qui est utilisée car elle est non bloquante. Si la structure est utilisée par une autre tâche, le portHandler ne peut pas stocker le message reçu dans la structure. Il serait possible d'oublier le message puisque celui-ci sera renvoyé quelques instants plus tard. Cependant, nous avons imaginé utiliser une structure temporaire dans laquelle sont stockés les messages qui n'ont pû être ajoutés à la liste des messages reçus. Si la structure des messages reçus est utilisée, le portHandler stocke le message dans la liste temporaire, Sinon, il recopie le contenu de la structure temporaire dans la liste des messages reçus et ajoute le message reçu en fin de liste.

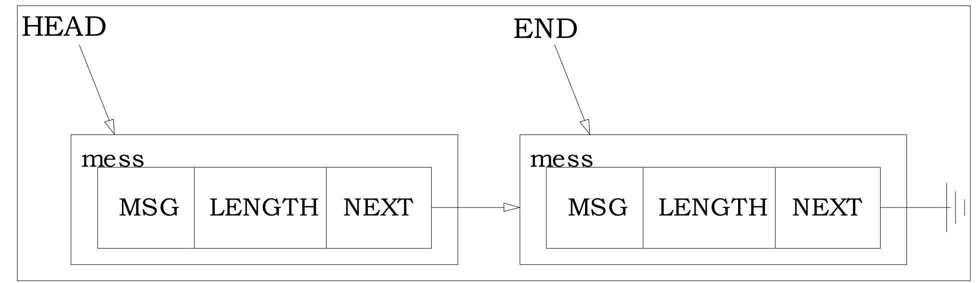


Figure 5.44 : Représentation de la structure des messages reçus.

Pour traiter les listes de messages reçus et de messages à envoyer, une tâche appelée `thread_ab` est créée. La gestion des messages reçus se fait de la manière suivante. La tâche consulte l'en-tête du message pour déterminer le type (acquittement ou non), le bit et la source.

Si le message n'est pas un acquittement, un acquittement est transmis à la station d'envoi. La figure 5.45 présente le format d'un acquittement qui découle de l'en-tête du message. Le bit d'envoi est recopié, les identificateurs de source et de destination sont inversés et le bit d'acquittement est mis à 1.

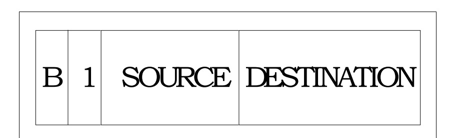


Figure 5.45 : Format d'un acquittement ab.

Si le bit du message est identique au bit de réception correspondant au site source, le message est libéré de son en-tête et transmis au portHandler de l'utilisateur comme présenté à la figure 5.46. Sinon, le message est oublié.

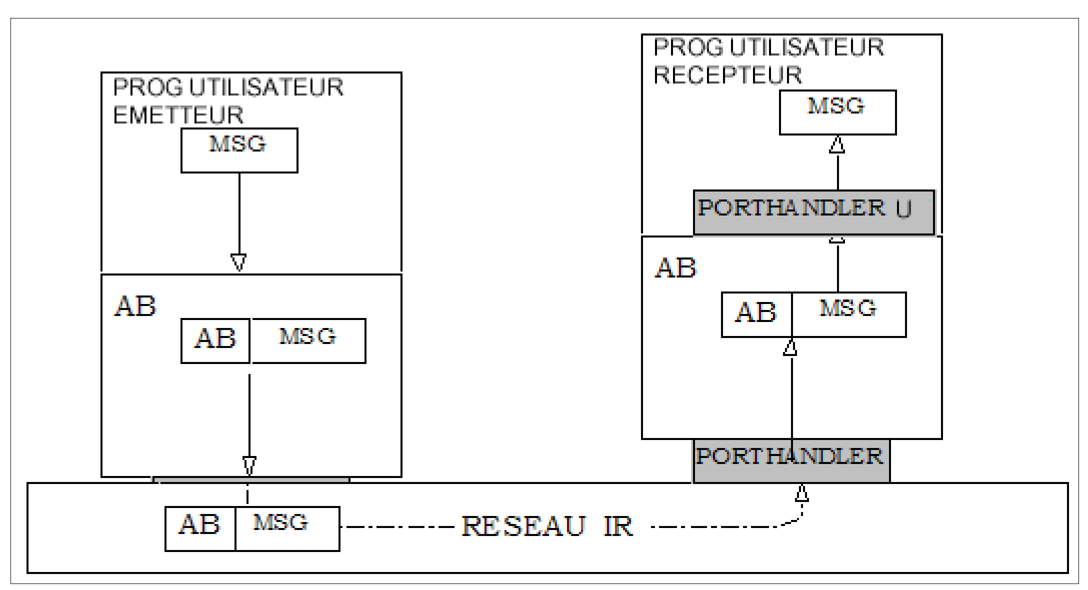


Figure 5.46 : Parcours d'un message envoyé.

Si le message reçu est un acquittement, la tâche parcourt la liste des sites à la recherche de l'élément correspondant au site source et compare le bit du message avec le bit d'envoi. S'ils ne coïncident pas, le premier message de la liste des messages du site est envoyé. Sinon, cela signifie que le message envoyé a bien été reçu et que celui-ci peut être retiré de la liste des messages à envoyer.

Le deuxième rôle de thread_ab est de parcourir la liste des sites et d'envoyer tous les premiers messages de chaque liste de messages. La gestion des temporisateurs a été quelque peu simplifiée. En effet, tester l'intervalle de temps écoulé depuis l'envoi d'un message et déterminer une valeur correcte pour le temporisateur nous a semblé trop spécifique. Nous nous sommes donc simplement assuré qu'un certain temps s'écoule entre deux envois successifs du même message. En pratique, la tâche ab va exécuter simultanément les deux fonctions qui lui sont attribuées et attendre un temps inversément proportionnel aux nombres de messages dans la liste des messages reçus. Le pseudo code de la tâche est repris à la figure 5.47.

```
    thread_ab {
         while(true) {
              traitement des messages reçus
              traitement des messages à envoyer
              msleep(10/(taille_liste_msg_reçu+1));
         }
   }
```

Figure 5.47 : Pseudo code de la tâche ab.

En résumé, trois éléments sont nécessaires pour l'implémentation du protocole :
- une fonction d'envoi appelée ab_write,
- un gestionnaire des messages reçus appelé ab_portHandler,
- et une tâche continue appelée thread_ab.

A cela, il faut ajouter les structures de données suivantes :
- une liste de messages à envoyer,
- et une liste de messages reçus.

Pour permettre l’usage du protocole, il est nécessaire de l’initialiser avec la fonction init_ab_protocol() et de lui fournir en paramètre un pointeur vers le portHandler de l’utilisateur ainsi que l’identificateur du site. Celui-çi est nécessaire pour la conception des en-têtes et la sélection des messages reçus par le ab_portHandler. Cette fonction initialisera les différentes structures, les sémaphores ainsi que la tâche thread_ab.

Le code du protocole de communication est disponible en *Annexe E*.

Chapitre VI

Etude de cas : chaîne de montage

Dans ce chapitre, nous étudierons le design d'un contrôleur d'une chaîne de montage afin d'illustrer les concepts de dSL et d'évaluer ce langage. Ensuite, nous parcourrons les principaux problèmes posés par la gestion de ce système. De plus, nous détaillerons quelques aspects liés à sa modélisation en Lego-Mindstorms pour terminer par son implémentation dSL et son évaluation.

1. La description du système

Le système consiste en un contrôleur d'une chaîne de montage. Celle-ci est composée de plusieurs tapis roulants et d'un ascenceur. Le principe de cette application est de faire défiler des colis du début à la fin des tapis roulants. Ces colis sont remplis au fur et à mesure de leur parcours. A chaque étape de construction, le colis est arrêté et transformé. Par exemple, il peut s'agir de boîtes en carton remplies suivant une commande. A chaque arrêt, un certain nombre de livres est ajouté dans la caisse. En début de chaîne, les colis peuvent arriver sur deux tapis. De manière similaire, deux tapis de sortie (correspondant à deux types de colis différents) ont été disposés à la fin de la chaîne. De plus, un ascenceur est introduit au milieu de celle-ci pour transmettre les colis du tapis de niveau inférieur à la suite de la chaîne. Les figures 6.1 et 6.2 représentent un schéma de la chaîne de montage vu respectivement de profil et de haut.

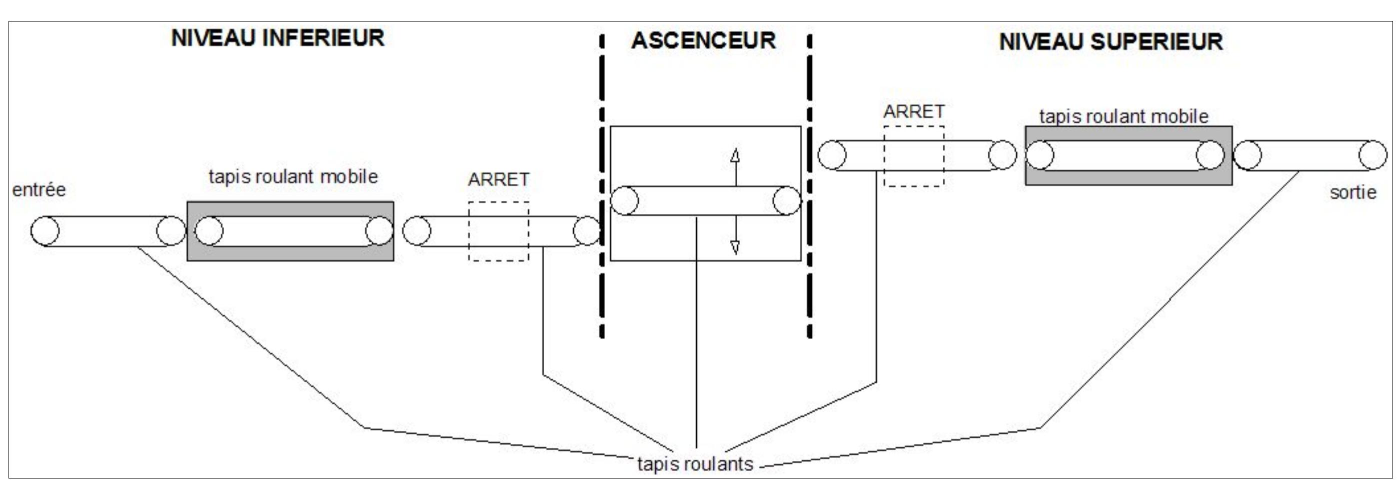


Figure 6.1 : Schéma de la chaîne de montage (vue de profil).

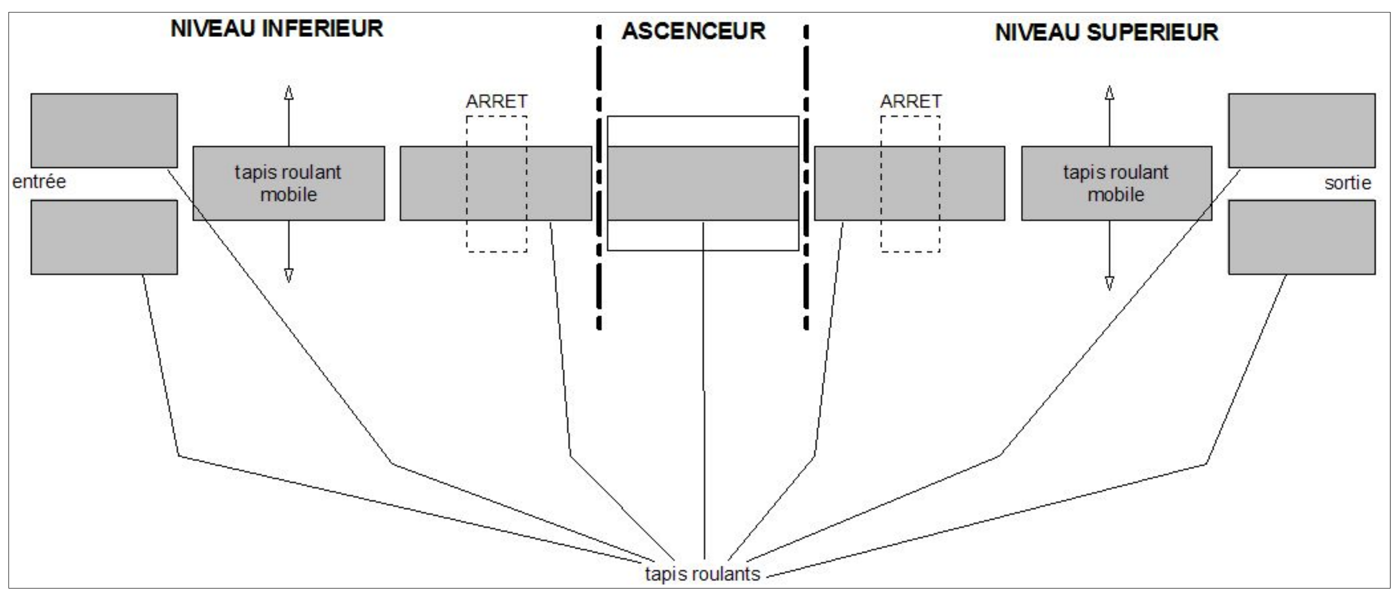


Figure 6.2 : Schéma de la chaîne de montage (vue de haut).

2. Les difficultés

2.1. Lancement et arrêt des tapis

La gestion de plusieurs tapis roulants consécutifs nécessite une attention particulière. En effet, le lancement et l'arrêt des tapis doivent être effectués avec précaution. En règle générale, en aucun cas un tapis ne peut être en marche alors que le suivant est à l'arrêt. Dans ce cas, un colis pourrait alors se retrouver coincé et abîmé. Les tapis pourraient aussi être endommagés. Pour ces raisons de sécurité et de sûreté, l'ordre de mise en route et d'arrêt des tapis doit être défini précisément. La mise en route se fait du dernier tapis au premier et inversément en ce qui concerne l'arrêt. Toute opération où l'ordre est important nécessite l'emploi d'une séquence.

2.2. Exclusion mutuelle des tapis d'entrée

La notion d'exclusion mutuelle entre deux opérations désigne le fait que l'une empêche l'autre. En d'autres termes, les deux opérations ne peuvent être effectuées en même temps. L'exclusion mutuelle résulte souvent de l'utilisation d'une ressource commune. Il en est question pour les deux tapis d'entrée. En effet, ils utilisent chacun le tapis roulant mobile pour ramener les colis sur le tapis principal. Lorsqu'un colis se présente à l'extrémité d'un des tapis d'entrée, celui-ci s'arrête et demande au tapis mobile de venir chercher le colis. Le tapis se remet en route lorsqu'il est sûr que le tapis mobile est situé en face de lui et qu'il peut donc lui transmettre son colis.

2.3. Section critique de l'ascenceur et du tapis de sortie

L'ascenceur est un élément intéressant de la chaîne de montage. Le tapis inférieur doit attendre sa disponibilité, c'est-à-dire que l'ascenceur soit en bas, avant de faire avancer le colis à l'intérieur de celui-ci. Idem en ce qui concerne le tapis roulant mobile de la sortie.

2.4. Communications inter site

De par les disponibilités limitées offertes par les briques Lego (trois entrées et trois sorties), chacune des briques sera chargée d'un élément particulier de la chaîne de montage. Un très grand nombre de messages de communication vont donc être transmis entre ces briques pour la bonne mise en œuvre du système. L'utilisation du protocole de communication fiable permet d'être certain de la fiabilité des messages. Cela dit, nous veillerons également à disposer les briques Lego d'une manière telle que celles qui devront s'échanger des messages ne soient pas à l'opposé l'une de l'autre. Si tel est le cas, il est possible d'introduire des miroirs dans la construction pour relayer les communications infrarouges.

3. Modélisation Lego-Mindstorms du système

En vue de réaliser le système en Lego-Mindstorms, un certain nombre de simplifications ont été apportées à l'idée de base de la chaîne de montage. Celles-ci résident essentiellement dans la modélisation des colis. Ceux-ci sont de simples briques Lego. Aucune opération sur les briques n'est effectuée lors des étapes de construction du colis. Un simple temps d'arrêt est marqué.

Par la suite, nous allons détailler la conception des différents éléments de la chaîne de montage, à savoir le tapis roulant, le tapis roulant mobile et l'ascenceur. En ce qui concerne les tapis, ils sont constitués d'une chenille placée sur deux roulements dont l'un des deux est actionné par un moteur. Un capteur de lumière permet de savoir lorsqu'un colis se trouve en fin de tapis. L'*annexe F* de ce texte expose la manière dont ces tapis roulants ont été construits.

En ce qui concerne les tapis roulants mobiles, ils consistent en un simple tapis roulant placé sur un chariot mobile. L'*annexe H* présente comment ce chariot mobile a été conçu. Il s'agit d'un chariot rotatif. Le moteur du chariot va actionner un engrenage qui va permettre de le faire pivoter. Des capteurs de toucher permettent de savoir lorsque le tapis est arrivé à une position correcte. Nous avons simplifié la gestion du chariot mobile en plaçant le tapis principal face à l'un des deux tapis d'entrée. Ainsi, il n'y a que deux positions possibles : soit en position 1, c'est-à-dire dans l'axe du premier tapis d'entrée et du tapis principal; soit en position 2, c'est-à-dire perpendiculairement à l'axe du tapis principal, face au deuxième tapis d'entrée.

Pour finir, l'ascenceur est l'élément le plus complexe de la chaîne. Le principe de sa construction est repris à l'*Annexe I.* Il consiste en une plate-forme sur laquelle deux moteurs actionnent deux axes terminés par deux engrenages chacun. Ces engrenages sont les points d'appui de la plate-forme et permettent de faire monter et descendre l'ascenceur. Le principe de construction de la cage d'ascenceur est précisé en *Annexe J.* Deux capteurs de toucher permettent de savoir lorsque l'ascenceur est arrivé en haut ou en bas. Bien entendu, pour assurer la continuité du tapis, un tapis roulant est disposé sur la plate-forme de l'ascenceur.

Avant de passer à l'implémentation dSL de la chaîne de montage, nous allons déterminer le nombre de briques Lego-Mindstorms nécessaires. La figure 6.3 récapitule les dispositifs d'entrée/sortie Lego-Mindstroms nécessaires pour la réalisation des différents éléments constituant le système. Chaque tapis roulant nécessite une sortie (le moteur) et une entrée (le capteur de lumière). Les chariots mobiles nécessitent chacun une sortie (le moteur du chariot) et deux entrées supplémentaires (les capteurs de toucher). En ce qui concerne l'ascenceur, il requiert une sortie (les deux moteurs qui seront connectés ensemble) et deux entrées (les capteurs de toucher). La figure 6.4 reprend l'ensemble des entrées et sorties nécessaires pour la réalisation de la chaîne de montage

ainsi que la distribution utilisée. Au total, six briques sont nécessaires. La première contrôle les deux tapis roulants d'entrée. La seconde se charge du tapis roulant mobile inférieur. La troisième s'occupe des deux tapis roulants intermédiaires. La quatrième est responsable de l'ascenceur, la cinquième du chariot mobile supérieur et la dernière des deux tapis roulants de sortie.

	Entrée		***Sortie***	
	Nombre	Détails	Nombre	Détails
Tapis roulant	1	capteur de lumière	1	moteur
Chariot mobile	2	capteurs de toucher	1	moteur
Ascenceur	2	capteurs de toucher	2	moteurs

Figure 6.3 : Récapitulatif des entrées/sorties par élément du système.

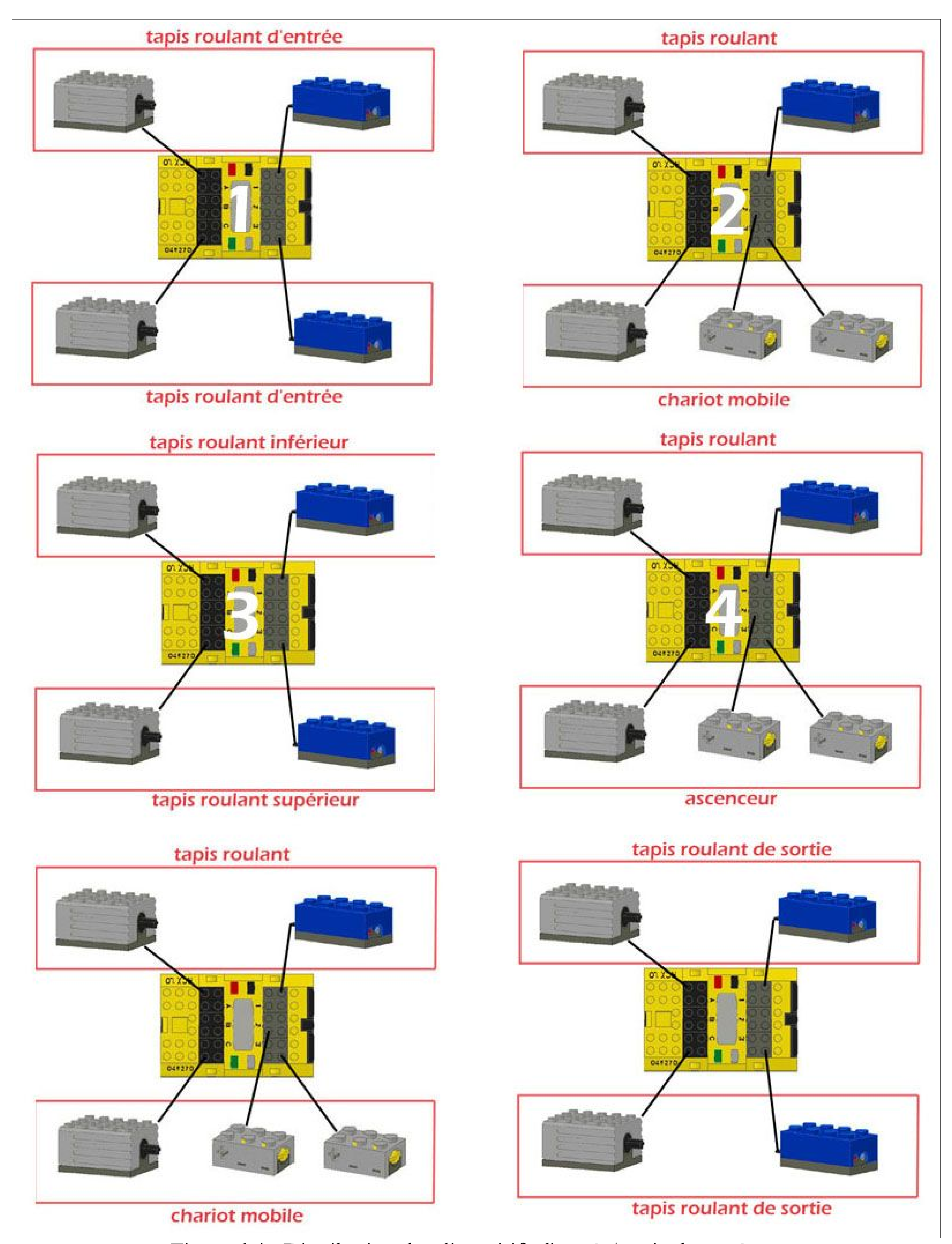


Figure 6.4 : Distribution des dispositifs d'entrée/sortie du système.

4. Implémentation dSL du système

Dans ce paragraphe, nous détaillerons l'implémentation des différents éléments du système, à savoir les tapis roulants, l'ascenceur et les chariots mobiles. Pour chaque élément, on peut définir une classe contenant les méthodes de manipulation et les règles de base à respecter. Ensuite, nous étudierons les liens établis entre ces différents éléments, c'est-à-dire les messages échangés entre les sites pour assurer le bon fonctionnement du système. Pour finir, nous détaillerons la distribution des variables globales sur les cinq sites constituant le système (cf. *3. Modélisation Lego-Mindstorms du système*).

4.1. Les tapis

```
CLASS conveyor_belt
   motor : INT;
   light : INT;
   free : BOOL;
END_CLASS

METHOD conveyor_belt::go()
   self.motor := 20;
END_METHOD

METHOD conveyor_belt::stop()
   self.motor := 0;
END_METHOD
```

Figure 6.5 : Code dSL de la classe conveyor_belt.

Le code de base de la classe conveyor_belt est repris dans la figure 6.5. Chaque tapis est constitué d'un moteur et d'un capteur de lumière. Les méthodes de manipulation sont la mise en route et l'arrêt du tapis. La vitesse des moteurs a été définie de manière empirique. La variable `free` est utilisée pour indiquer si une brique est sur le tapis ou non.

4.2. L'ascenceur

```
CLASS elevator
   motor : INT;
   touch_up : INT;
   touch_down : INT;
   rotor : conveyor_belt;
   state : BOOL; (*TRUE:running, FALSE:stopped*)
   position : BOOL; (*TRUE:up, FALSE:down*)
   free : BOOL;
END_CLASS

METHOD elevator::up()
   IF(touch_up<0) THEN
      self.state := FALSE;
      self.rotor<-stop();
      self.motor := 20;
   END_IF;
END_METHOD

METHOD elevator::down()
   IF(touch_down<0) THEN
      self.state := FALSE;
      self.rotor<-stop();
      self.motor := -2;
   END_IF;
END_METHOD

METHOD elevator::stop()
   self.motor := 0;
   self.state := TRUE;
END_METHOD

WHEN IN elevator touch_up > 0 THEN
   self.position := TRUE;
   self<-stop();
END_WHEN

WHEN IN elevator touch_down > 0 THEN
   self.position := FALSE;
   self<-stop();
END_WHEN
```

Figure 6.6 : Code dSL de la classe elevator.

Un ascenceur est constitué d'un moteur, de deux capteurs situés en haut et en bas de la cage et d'un tapis roulant. Les méthodes de base consistent à monter, descendre et arrêter l'ascenceur. Les règles de base sont que lorsque l'ascenceur arrive en haut ou en bas, les moteurs s'arrêtent. Nous avons ajouté deux variables internes à l'ascenceur représentant son état et sa position. L'ascenceur est en position bas si position est égal à FALSE et si l'état est égal à TRUE. L'ascenceur est en position haut si position est égal à TRUE et si l'état est égal à TRUE. Lorsque l'état est égal à FALSE, l'ascenceur est en mouvement. Lorsque l'ascenceur est en mouvement, il n'est pas permis que le tapis soit en mouvement. La figure 6.6 présente le code de la classe elevator. La variable `free` indique si l'ascenceur est en cours d'utilisation ou non.

4.3. Les chariots mobiles

```
CLASS moving_cart
   motor : INT;
   touch_one : INT;
   touch_two : INT;
   position : BOOL; (*TRUE:one, FALSE:two*)
   state : BOOL; (*TRUE:running, FALSE:stopped*)
   free : BOOL;
END_CLASS

METHOD moving_cart::go_in_one()
   IF(self.touch_one<0) THEN
      self.state := FALSE;
      self.rotor<-stop();
      self.motor := -30;
   END_IF;
END_METHOD

METHOD moving_cart::go_in_two()
   IF(self.touch_two<0) THEN
      self.state := FALSE;
      self.rotor.motor:=0;
      self.motor := 30;
   END_IF;
END_METHOD

METHOD moving_cart::stop()
   self.motor := 0;
   self.state := TRUE;
END_METHOD

WHEN IN moving_cart self.touch_one > 0 THEN
   self.position := TRUE;
   self<-stop();
END_WHEN

WHEN IN moving_cart self.touch_two > 0 THEN
   self.position := FALSE;
   self<-stop();
END_WHEN
```

Figure 6.7 : Code dSL de la classe moving_cart.

Chaque chariot mobile est composé d'un moteur, de deux capteurs (un pour chaque position du chariot) et d'un tapis roulant. Un chariot possède aussi deux variables indiquant son état et sa position. Les méthodes de base consistent à amener le chariot en position 1 (`go_in_one()`) ou 2 (`go_in_two()`), ou à l'arrêter. Les règles de base sont que lorsque le chariot arrive dans une position ou dans l'autre, il s'arrête. Le code de la classe est repris à la figure 6.7. L'utilité de la variable `free` sera expliquée dans le paragraphe suivant.

4.4. Les liens entre les éléments de la chaîne

Le système implémenté est un système réactif. Il réagit à un certain nombre d'événements. Ces événements sont liés aux capteurs de lumière qui détectent la présence d'une brique sur un tapis roulant.

Avant de commencer à les détailler, nous allons évoquer un problème auquel nous avons dû faire face dans l'implémentation. Les capteurs de lumière manifestent une très grande sensibilité aux variations de luminosité. La valeur seuil, c'est-à-dire la valeur (relevée par le capteur) sous laquelle on peut être sûr qu'une brique est en face du capteur est très difficile à déterminer de manière précise. Elle peut varier en fonction de l'inclinaison des tapis par rapport à la lumière ou même d'un instant de la journée à un autre. Nous avons tenté de créer des conditions optimales afin que l'environnement n'influence pas le comportement du système. Pour cela, nous avons construit des ponts recouvrants les tapis roulants. Ces ponts sont construits en briques blanches. Nous utilisons alors des colis de couleur noire pour maximiser la différence de relevé. Le seuil a ensuite été défini à la valeur 39 de manière empirique. Remarquons l'usage d'un « `#define TRESHOLD` *`valeur`* » qui permet de spécifier la valeur du seuil en début de code. Pour modifier cette valeur, il ne suffit que d'une modification dans le code. En cas de define, le pré-processeur traduit automatiquement les TRESHOLD en la valeur définie en début de code.

Pour rappel, un événement dSL est défini par une condition et par une suite d'instructions de traitement. Cette construction est soumise à la contrainte atomique. Deux mécanismes permettent de contourner cette contrainte : les tildes et les LAUNCH. Une certaine dualité existe entre ces deux mécanismes. Il a été prouvé dans **[DMM03(2)]** que toute instruction LAUNCH et que toute séquence peuvent être traduites de manière équivalente en un code n'utilisant que des WHEN et des tildes. Cela dit, en pratique, il est préférable de n'utiliser un tilde uniquement si la valeur de la variable est utilisée dans l'évaluation d'une expression telle que celle d'une assignation. Nous aurions pu par exemple tilder la valeur d'un capteur de lumière dans l'expression d'une condition d'un événement distribuée sur le site du chariot mobile par exemple. À chaque fois que ce capteur va relever une valeur différente de la précédente, un message sera envoyé. En pratique, c'est souvent le cas à cause de l'extrême sensibilité des capteurs de lumière. Même si rien ne passe devant eux, ils relèvent de petites variations de luminosité. Dans le cas qui nous intéresse ici, il est donc préférable d'utiliser des mécanismes LAUNCH. Un seul message est envoyé dès que l'exécution d'une séquence est nécessaire.

Nous avons donc défini un certain nombre de séquences qui prendront en charge le traitement des briques. Puisqu'une seule instance de séquence ne peut être exécutée à tout instant, nous avons veillé à découper le traitement d'une brique de l'entrée à la sortie en le plus grand nombre de séquences possibles et, ainsi, à en limiter la taille. La taille minimale d'une séquence correspond à la prise en charge d'une brique sur une section où il ne peut y avoir qu'une seule brique en cours de traitement. Par exemple, le chariot mobile ou l'ascenceur. Les séquences sont donc définies par rapport à la découpe qui peut être faite de la topologie de la chaîne. La séquence `t1_to_t3()` prend en charge la brique du tapis 1 au tapis 3 en passant par le chariot mobile. Idem pour `t2_to_t3()`. `t3_to_t4()` fait passer la brique du tapis 3 au tapis 4 en tranversant l'ascenceur. `t4_to_t5()` et `t4_to_t6()` se chargent d'amener une brique du tapis 4 vers les tapis 5 ou 6. Les deux premières séquences sont mutuellement exclusives. Elles utilisent en effet la même ressource. Nous avons donc défini un jeton correspondant à l'accès au premier chariot mobile. Ce jeton est une variable booléenne `free` mise à vrai lorsque la ressource chariot mobile est disponible et mise à faux lorsque la ressource est utilisée. La modification du jeton doit se faire en tout début et en toute fin de l'exécution des séquences `t1_to_t3()` et `t2_to_t3()`. Elle lance aussi la séquence suivante, c'est -à-dire la séquence `t3_to_t4()`.

Nous avons aussi défini deux variables `ask_one` et `ask_two` qui indiquent si une brique est en attente sur le tapis 1 ou sur le tapis 2. Les conditions des événements sont donc définies sur ces variables. Les variables `ask_one` et `ask_two` sont mises à jour grâce aux événements de la figure 6.8 qui permettent également d'arrêter les tapis lorsqu'une brique se présente face aux capteurs.

```
    WHEN ( t_2.light < TRESHOLD AND ask_two==FALSE ) THEN
       t_2<-stop();
       ask_two:=TRUE;
    END_WHEN

    WHEN ( t_2.light >= TRESHOLD AND ask_two==TRUE ) THEN
       ask_two:=FALSE;
    END_WHEN

    WHEN ( t_1.light < TRESHOLD AND ask_one==FALSE ) THEN
       t_1<-stop();
       ask_one:=TRUE;
    END_WHEN

    WHEN ( t_1.light >= TRESHOLD AND ask_one==TRUE ) THEN
       ask_one:=FALSE;
    END_WHEN
```

Figure 6.8 : Code des événements d'arrêt des tapis d'entrée et de mise à jour des variables `ask_one` et `ask_two`.

L'utilité des variables `ask_one` et `ask_two` réside dans l'observation suivante. Sémantiquement, lorsqu'une variable est modifiée, il faut évaluer les conditions des événements utilisant cette variable. Nous avons vu que les valeurs relevées par les capteurs de lumière sont très variables. Les variables `ask_one` et `ask_two` permettent donc de limiter le nombre de conditions évaluées aux quatre événements mentionnés dans la figure 6.8. Les conditions des événements de lancement des séquences sont donc définies sur les valeurs `ask_one` et `ask_two` mais aussi sur `c_1.free`, l'état du chariot mobile. Lorsqu'une brique est présente sur un tapis d'entrée et que le chariot est disponible, la séquence de traitement correspondante est lancée. Pendant l'exécution de la séquence, les briques arrivant sur les tapis d'entrée sont mises en attente. Il est donc possible qu'une brique se trouve sur chacun des tapis lors de la libération du chariot. Il a donc fallu prévoir un mécanisme de décision de la brique à traiter dans ce cas. Pour cela, nous avons utilisé une variable `last_in` qui stocke la dernière décision prise. La brique traitée est celle située sur le tapis qui n'avait pas été traité précédemment. Nous obtenons donc les trois événements de la figure 6.9.

```
    WHEN (ask_one==TRUE AND ask_two==FALSE AND c_1.free==TRUE) THEN
       c_1.free:=FALSE;   (*take control of the moving cart*)
       LAUNCH t1_to_t3();
       last_in:=TRUE;
    END_WHEN

    WHEN (ask_one==FALSE AND ask_two==TRUE AND c_1.free==TRUE) THEN
       c_1.free:=FALSE;   (*take control of the moving cart*)
       LAUNCH t2_to_t3();
       last_in:=FALSE;
    END_WHEN

    WHEN (ask_one==TRUE AND ask_two==TRUE AND c_1.free==TRUE) THEN
       c_1.free:=FALSE;   (*take control of the moving cart*)
       IF(last_in==FALSE) THEN
          LAUNCH t1_to_t3();
          last_in:=TRUE;
       ELSE
          LAUNCH t2_to_t3();
          last_in:=FALSE;
       ENDIF;
    END_WHEN
```

Figure 6.9 : Code des événements de lancement des séquences `t1_to_t3()` et `t2_to_t3()`.

En tant que tapis d'entrée, les tapis 1 et 2 doivent tourner tant qu'il n'y a pas de brique en face de leur capteur. Les événements de la figure 6.10 assurent cela.

```
WHEN ask_on == FALSE THEN
   t_1<-go();
END_WHEN

WHEN ask_two == FALSE THEN
   t_2<-go();
END_WHEN
```

Figure 6.10 : Code des événements de mise en route des tapis d'entrée.

Après avoir défini les événements, il est nécessaire de détailler les séquences. Lorsqu'une brique est en face d'un des deux capteurs des tapis d'entrée, il faut

1. amener le chariot mobile en face du tapis d'entrée,
2. lancer les tapis jusqu'à ce que le colis soit sur le chariot mobile,
3. amener le chariot mobile face au tapis intermédiaire,
4. lancer les tapis jusqu'à ce que la brique soit sur le tapis intermédiaire,
5. lancer la séquence suivante à condition que l'ascenceur soit libre.

Tout ce traitement ne peut être fait de manière naturelle, efficace et sûre que sous forme d'une séquence détaillée dans la figure 6.11. De plus, une séquence permet d'utiliser des instructions d'attente.

Remarquons qu'il est nécessaire d'attendre que l'ascenceur soit libre avant de lancer la séquence `t3_to_t4()` car il ne peut y avoir plus d'une instance d'une séquence en cours d'exécution.

```
(***carry brick from t_2 to t_3***)
SEQUENCE t2_to_t3()

   (*get the moving cart in position 2*)
   c_1<-go_in_two();
   wait(c_1.state==TRUE AND c_1.position==FALSE);

   (*launch the two conveyer belts*)
   c_1.rotor<-go();
   LAUNCH t_2<-go();

   (*brick on conveyer belt of conveyer belt*)
   wait(c_1.rotor.light<TRESHOLD);

   (*get the moving cart in position 2*)
   LAUNCH c_1<-go_in_one();
   wait(c_1.state==TRUE AND c_1.position==TRUE);

   (*wait the next conv. belt to be free*)
   wait(t_3.free==TRUE);
   t_3.free:=FALSE;

   (*launch the two conveyer belts*)
   t_3<-go();
   c_1.rotor<-go();

   (*brick on conveyer belt 3*)
   wait(t_3.light<TRESHOLD);
   t_3<-stop();
   c_1.rotor<-stop();

   (*wait the elevator to be free*)
   wait(a_1.free==TRUE);
   a_1.free:=FALSE;

   (*launch the next sequence*)
   LAUNCH t3_to_t4();

   (*free the moving cart*)
   c_1.free:=TRUE;
END_SEQUENCE
```

Figure 6.11 : Code de la séquence `t2_to_t3()`.

Le traitement est similaire pour le tapis mobile supérieur et pour l'ascenceur. La figure 6.12 reprend le code de la séquence `t3_to_t4()`. Elle lance une des séquences suivantes, à savoir une fois `t4_to_t5()` et une fois `t4_to_t6()` pour transférer la brique sur un des tapis de sortie. La variable `last_out` est utilisée pour stocker la dernière décision prise par rapport au tapis de sortie.

```
(***carry brick from t_3 to t_4***)
SEQUENCE t3_to_t4()

   (*wait the elevator to be in position down*)
   wait(a_1.position==FALSE AND a_1.state==TRUE);

   (*launch the two conveyer belts*)
   a_1.rotor<-go();
   t_3<-go();

   (*brick on conveyer belt of elevator*)
   wait(a_1.rotor.light<TRESHOLD);
   a_1.rotor<-stop();
   t_3<-stop();

   (*free the conveyer belt 3*)
   t_3.free:=TRUE;

   (*get the elevator in position up*)
   a_1<-up();
   wait(a_1.state==TRUE AND a_1.position==TRUE);

   (*launch the two conveyer belts*)
   t_4<-go();
   a_1.rotor<-go();
   wait(t_4.light<TRESHOLD);

   t_4<-stop();
   a_1.rotor<-stop();

   (*bring back the elevator in position down*)
   LAUNCH a_1<-down(); (*asynchron call*)

   (*wait the moving cart to be free*)
   wait (c_2.free==TRUE);
   c_2.free:=FALSE;

   (*launch the next sequence*)
   IF (last_out==TRUE) THEN
      last_out:=FALSE;
      LAUNCH t4_to_t5();
   ELSE
      last_out==TRUE;
      LAUNCH t4_to_t6();
   END_IF;

   (*free the elevator*)
   a_1.free:=TRUE;
END_SEQUENCE
```

Figure 6.12 : Code de la séquence `t3_to_t4()`.

Bien entendu, une séquence d'initialisation est définie. Elle intialise toutes les variables utilisées dans le code, et notamment les variables `free`. Le code de tout le système est reprise en *Annexe K*.

5. Évaluation

L'usage de dSL pour implémenter un système LegOS contribue à réduire de manière évidente le travail du programmeur tant du point de vue de la conception que de la taille du code à générer, et en particulier si on considère des systèmes distribués sur plusieurs briques RCX.

Le langage dSL est un langage spécifique pour la modélisation des systèmes réactifs. Il s'avère idéal pour implémenter des systèmes LegOS qui peuvent toujours être vus comme des systèmes réactifs. En effet, un programme LegOS réagit à des stimuli tels que la pression d'un bouton ou le changement de luminosité devant un capteur de lumière. Dans le paragraphe 4, nous avons pu voir la simplicité avec laquelle les éléments de la chaîne ont pu être implémentés en dSL.

D'autre part, dSL offre une vision centralisée du système. Le compilateur génère automatiquement tout ce qui concerne la distribution du système tant du point de vue des messages échangés que des traitements distribués. Si nous voulions implémenter un système distribué en C pour LegOS, il serait nécessaire de prévoir toute une série de choses telles que la syntaxe des messages échangés et les fonctions de conception, d'envoi, de réception et de traitement des messages. De plus, afin de garantir un comportement distribué correct, le programmeur devrait générer lui-même toutes les communications nécessaires (transfert de valeur, exécution de méthode, etc.). En outre, tout ce qui concerne la mise en œuvre de tâches est gérée par le compilateur.

La simplicité d'implémentation est manifeste si on compare le nombre d'instructions à produire pour programmer un système. Dans tous les exemples que nous avons implémenté au cours du développement du compilateur, nous avons pu constater en général une diminution du nombre d'instructions entre le programme dSL et le code LegOS équivalent. En ce qui concerne l'étude de cas, si nous comparons la taille du fichier dSL (environ 400 lignes) avec une estimation du nombre d'instructions qui auraient été nécessaires pour implémenter le système directement en C pour LegOS (entre 200 et 400 lignes suivant le site, pour 2000 lignes au total), nous pouvons estimer le gain à environ **75%**. De plus, en cas par exemple de changement de position d'un capteur, les modifications à apporter au programme dSL sont minimes.

Pour finir, les principales constructions LegOS sont implémentables en dSL. Par exemple, la notion d'événement LegOS est facilement implémentable sous forme de WHEN ou de WAIT.

Bien entendu, l'utilisation de dSL pour implémenter un système LegOS a des désavantages. Le premier est la non-modularité de dSL. Le respect de la contrainte atomique impose parfois quelques difficultés de programmation. De plus, l'ensemble des instructions disponibles en dSL ne recouvrent pas l'entièreté des possibilités de LegOS. Par exemple, aucune instruction de production de son n'est disponible en dSL. Cela dit, au vu de la bonne lisibilité du code produit par le compilateur, il est aisé pour le programmeur de retoucher le code C en y ajoutant des instructions plus spécifiques. dSL est aussi limité par rapport à l'utilisation de structure de donnée particulière (string, pointeur, etc.).

Chapitre VII

Conclusions

1. Résumé

Dans ce texte, nous avons défini de la manière la plus complète possible le langage dSL, un langage de design de systèmes de contrôle distribué. Nous avons extrait les principaux avantages de dSL par rapport à son ancêtre SL et à d'autres approches comme les langages synchrones. Une présentation de la syntaxe et de la sémantique de dSL a été faite.

Nous avons par la suite présenté les Lego-Mindstorms qui ont été utilisés comme laboratoire d'expérimentation du langage et nous avons motivé le choix de LegOS parmi les systèmes d'exploitation disponibles.

Ensuite, nous avons étudié la génération de code de dSL vers LegOS et nous l'avons illustrée par une série d'exemples. Nous avons réalisé un compilateur qui génère automatiquement le code de tous les sites du système.

Nous avons également présenté le mode de fonctionnement et l'implémentation du protocole de communication fiable que nous avons implémenté.

Pour finir, nous avons mis en œuvre un exemple concret de système de contrôle distribué, à savoir une chaîne de montage, tant du point de vue de l'implémentation dSL que de la construction Lego-Mindstorms. Cette étude de cas nous a permis d'une part d'évaluer le travail effectué sur le compilateur et d'autre part de valider l'approche dSL de manière pragmatique.

Les résultats de cette évaluation sont une réduction de l'effort de programmation du concepteur du système tant du point de vue de la conception que de la taille du code à produire. Ces facilités sont issues du fait que dSL permet de ne pas devoir gérer explicitement les aspects de communication entre les sites et de se concentrer sur les fonctionnalités du système. Il offre en outre une vision centralisée du système. Le compilateur que nous avons créé permet de simplifier la programmation des Lego-Mindstorms en utilisant le langage dSL pour modéliser des systèmes distribués sur plusieurs briques RCX. Il permet en outre de recourir aux Lego-Mindstorms afin de tester des implémentations dSL de systèmes réactifs distribués. Il serait d'ailleurs intéressant que le compilateur soit mis à disposition sur Internet.

2. Travaux futurs

Puisque les systèmes implémentés sont critiques, la spécification et la vérification des programmes dSL doit être étudiée. Cela peut être réalisé par l'utilisation de *Spin* comme introduit dans [**DMM03(2)**]. Une autre nécessité est de prouver qu'une distribution est correcte, c'est-à-dire que le programme distribué de manière transparente est conforme avec la sémantique du problème original.

Le langage dSL est toujours en plein développement. Il peut donc être raffiné par l'ajout de nouveaux types de données plus complexes (comme les pointeurs et les structures), de constructions supplémentaires (Programmation Orienté-Objet, sémaphores, etc.) ainsi que par des librairies d'automatisation qui réduiront la tâche du programmeur.

Signalons aussi qu'il reste quelques fonctionnalités dSL qui ne sont pas encore prises en compte par le compilateur, essentiellement tout ce qui concerne les vecteurs.

Bibliographie

ASU00 A. Aho, R. Sethi, J. Ullman, Compilers. Principles, techniques and tools, Ed. Dunod, 2000.

BGT00 D. Baum, M. Gasperi, R. Hempel et L. Villa, Extreme Mindstorms. An advanced guide to Lego Mindstorms, Ed. Apress, 2000.

Bau00 D. Baum, Definitive Guide To Lego-Mindstorms, Ed. Apress, 2000.

BGJ91 A. Benveniste, P. Le Guernic et C. Jacquemot, Synchronous programming with events and relations: the Signal Language and its semantics,

Dans "*Science of Computer Programming"*, 16(2):103-149, Sept. 1991.

Ber98 Gérard Berry, The Foundations of Esterel,

Dans " *Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte*", MIT Press, 1998.

BD91 Frédéric Boussinot et Robert De Simone, The Esterel Language,

Dans "Another Look at Real Time Programming", Proc. of the IEEE, vol. 79, pp 1293-1304, 1991.

CPHP87 P. Caspi, D. Pilaud, N. Halbwachs et J. A. Plaice, Lustre: a declarative language for programming synchronous systems,

Dans "*ACM Symp. on Principles of Programming Languages (POPL)"*, Munich, 1987.

Dew02 Bram De Wachter, Code distribution in the dSL environment for the synthesis of industrial process control, Bruxelles, 2002.

DMM03 Bram De Wachter, Thierry Massart, Cédric Meuter, dSL : An environment with automatic code distribution for industrial control systems, Bruxelles, 2003.

DMM03(2) Bram De Wachter, Alexandre Genon, Thierry Massart et Cédric Meuter, dSL to design distributed industrial control systems and verify them with *Spin*, Bruxelles, 2003.

DMM04 Bram De Wachter, Thierry Massart et Cédric Meuter, dSL : An environnement with automatic code distribution for industrial control systems. Technical Report 512, Bruxelles, 2004.

HIT Hitachi Single-chip microcomputer H8/3297 Series Hardware Manual

LFS02 D. Laverde, G. Ferrari et J. Stuber, Programming Lego Mindstorms with Java, Ed. Syngress, 2002.

Nie00 S. Nielsson, Introduction to the LegOS kernel, Sept. 2000

NL91 B. Nizeberg et V. Lo. Distributed shared memory : A survey of issues and algorithms. IEEE Computer, vol.24, no.8, pp. 52-60, Aug. 1991.

TV02 A.S. Tanenbaum et M. Van Steen, Distributed Systems, principles and paradigms, Ed. Prentice Hall, 2002.

Liste des illustrations

Chapitre 2 - Le langage dSL

Chapitre 3 - Les Legos - Mindstorms

Chapitre 4 – Compilation et distribution

Chapitre 5 – Génération automatique du code LegOS pour Lego-Mindstorms

Chapitre 6 – Etude de cas : chaîne de montage

Annexe A

Grammaire dSL

root	::=	PARSE_DSL_PROGRAM dsl_program
	|	PARSE_LHSIDES localizable_lhside_list
dsl_program	::=	declaration_list
declaration_list	::=	{ declaration }
declaration	::=	class
	|	global_variables
	|	site
	|	when
	|	when_in
	|	method
	|	sequence
class	::=	**"CLASS"** ID variable_declaration_list **"END_CLASS"**
global_variables	::=	**"GLOBAL_VAR"** variable_declaration_list **"END_VAR"**
local_variables	::=	[**"LOCAL_VAR"** variable_declaration_list **"END_VAR"**]
variable_declaration_list	::=	{ variable_declaration ";" }
variable_declaration	::=	non_empty_id_list ":" type
non_empty_id_list	::=	ID { "," ID }
site	::=	**"SITE"** ID localization_list **"END_SITE"**
localization_list	::=	{ localization ";" }
localization	::=	kind localizable_lhside ":" NUMBER "." NUMBER "." NUMBER
localizable_lhside	::=	ID
	|	localizable_lhside "." ID
	|	localizable_lhside "[" constant "]"
localizable_lhside_list	::=	{ localizable_lhside }
kind	::=	**"INTERNAL"**
	|	**"INPUT"**
	|	**"OUTPUT"**
when	::=	**"WHEN"** rhside **"THEN"** local_variables instruction_list **"END_WHEN"**
when_in	::=	**"WHEN"** **"IN"** ID rhside **"THEN"** local_variables instruction_list **"END_WHEN"**

```
method                      ::= "METHOD" ID "::" ID "(" parameter_declaration_list ")"
                                local_variables instruction_list "END_METHOD"
sequence                    ::= "SEQUENCE" ID ( "(" parameter_declaration_list ")"
                                local_variables instruction_list "END_SEQUENCE" | "::" ID
                                "(" parameter_declaration_list ")" local_variables instruction_list
                                "END_SEQUENCE" )
parameter_declaration_list  ::= [ variable_declaration [ "," parameter_declaration_list ] ]
type                        ::= "LONG"
                              | "INT"
                              | "BOOL"
                              | ID
                              | "ARRAY" "[" NUMBER ":" NUMBER "]" "OF" type
instruction_list            ::= { instruction ";" }
instruction                 ::= assign
                              | wait
                              | if
                              | while
                              | call
assign                      ::= lhside ":=" rhside
wait                        ::= "WAIT" rhside
if                          ::= "IF" rhside "THEN" instruction_list else
else                        ::= [ "ELSE" instruction_list ] "END_IF"
while                       ::= "WHILE" rhside "DO" instruction_list "END_WHILE"
lhside                      ::= ID
                              | lhside "." ID
                              | lhside "[" rhside "]"
rhside                      ::= lhside
                              | constant
                              | "(" rhside ")"
                              | rhside "OR" rhside
                              | rhside "AND" rhside
                              | rhside "<" rhside
                              | rhside ">" rhside
                              | rhside "<=" rhside
                              | rhside ">=" rhside
                              | rhside "<>" rhside
                              | rhside "==" rhside
                              | rhside "MOD" rhside
                              | rhside "+" rhside
                              | rhside "−" rhside
                              | rhside "*" rhside
                              | rhside "/" rhside
                              | "NOT" rhside
                              | "−" rhside
                              | "IS_UNKNOWN" rhside
                              | "~" lhside
constant                    ::= "TRUE"
                              | "FALSE"
                              | NUMBER
```

call	::=	(lhside “< –” | LAUNCH lhside “< –” | LAUNCH) ID **“(”** rhside_list **“)”**
rhside_list	::=	[rhside [**“,”** rhside_list]]

[T] Means 1 at least one T.
{ T } Means T or ϵ
“T” is a literal.
T is a terminal.

Annexe B

Sémantique dSL

1 Equivalence relation

Definition 1 *Let $Var(e) : Expr \mapsto 2^{Id}$ be the function that maps an expression to the variables appearing in that expression :*

$$Var(e) = \begin{cases} \{id\} & e = id \\ Var(e_1) & e = (e_1), e = \neg e_1 \\ Var(e_1) \cup Var(e_2) & e = e_1 \vee e_2, e = e_1 \wedge e_2 \\ \phi & e = \top, e = \bot, e = \widetilde{id} \end{cases}$$

Let $Var(E) = \cup_{e \in E} Var(e)$.

Definition 2 *Let $Var(instr) : Instr \mapsto 2^{Id}$ be the function that maps an instruction to the variables that appear in the immediate constituents of that instruction :*

$$Var(i) = \begin{cases} \{id\} \cup Var(expr) & i = id := expr \\ Var(expr) & i = while\ expr\ do\ st_list\ end_while \\ Var(expr) & i = if\ expr\ then\ st_list\ end_if \end{cases}$$

Let $Var(I) = \cup_{i \in I} Var(i)$.

Definition 3 *Let $Expr(W)$ denote all expressions that appear in a given when W.*

Definition 4 *Let $\equiv_w$ be the equivalence relation between the whens of a $_dSL$ program resulting from the symmetric, reflexive and transitive closure of R_w, where*
$W_1 R_w W_2 \Leftrightarrow Var(Expr(W_1)) \cap Var(Expr(W_2)) \neq \phi$

Definition 5 *Let $\equiv_e$ be the equivalence relation between the expressions of a $_dSL$ program that satisfies the locality and access constraints, resulting from the symmetric, reflexive and transitive closure of R_e, where*
$expr_1 R_e expr_2 \Leftrightarrow (Var(expr_1) \cap Var(expr_2) \neq \phi) \vee (\exists W_1, W_2 : expr_1 \in Expr(W_1) \wedge expr_2 \in Expr(W_2) \wedge W_1 \equiv_w W_2) \vee (\exists instr : Var(expr_1) \cup Var(expr_2) \subseteq Var(instr))$

Definition 6 *Let $1, \cdots, N$ be the indices of the equivalence classes defined by $\equiv_e$, and let $i = 1, \cdots, N : Expr_i$ denote these classes.*

Definition 7 *Let $IVar_i$ be the input variables in $Var(Expr_i)$*

Definition 8 *Let $OVar_i$ be the output variables in $Var(Expr_i)$*

2 State

In this section we define the state that is used in structural operational semantics of a $_d$SL program, based on the equivalence classes defined in the previous section. A state is a 6-tuple :

- $\mathcal{V} : Var \times \{1, \cdots, N\} \mapsto \{\top, \bot\}$, a valuation for a given process of the global variables, inputs and outputs.
- $\mathcal{F} : \{1, \cdots, N\} \times \{1, \cdots, N\} \mapsto \Sigma^*$ where $\Sigma = Var \times \{\top, \bot\} \cup st_list$: The fifo channels that interconnect the different processes containing either variables and their values or statements to be executed.
- $\mathcal{S} : \{1, \cdots, N\} \mapsto 2^{st_list}$, the bodies of sequences that are sleeping in a given process.
- $\mathcal{A} : \{1, \cdots, N\} \mapsto 2^{st_list}$, the bodies of sequences that are active (awaiting execution) in a given process.
- $\mathcal{M} : \{1, \cdots, N\} \mapsto 2^{st_list}$, the bodies of sequences that are being transferred to a certain process.
- $\varphi : \{1, \cdots, N\} \mapsto \{Input, Msg, Process, Output\}$, the phase of each process, respectively sampling the inputs, handling messages from other processes, executing sequences and writing outputs.

We will use the notation $\mathcal{S}_i$ for $\mathcal{S}(i)$, (idem for $\mathcal{A}, \mathcal{M}$) and $\mathcal{F}_{ij}$ for $\mathcal{F}(i,j)$.

3 Transitions

3.1 Input

Input

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{?\{f : IVar_i \mapsto \{\top,\bot\}\}} < \cdots \| \overbrace{W_1; \cdots; W_n}^{i} \| \cdots, s' > \quad \text{if } S \models (\varphi_i = Input)$$

- $s_1 = s[\forall x \in IVar_i : \mathcal{V}(i)[\![x]\!] \mapsto f(x), \forall j \in \{1, \cdots, N\} \backslash \{i\} : \mathcal{F}_{ij} \mapsto \mathcal{F}_{ij} \cdot (x, f(x))]$
  The first part takes the inputs, and assigns them to the input variables of process i, while the second part sends the new values to all other processes
- $1, \cdots, k : W_k$ are the whens associated to process i
- $s' = s_1[\varphi_i \mapsto Msg; \mathcal{A}_i \mapsto \mathcal{S}_i; \mathcal{S}_i \mapsto \phi]$
  Input is done so pass to the process phase, and make all inactive sequence waiting for execution.

3.2 Handling messages

Read value

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{x := val_x}^{i} \| \cdots, s' >$$
$$\text{if } s \models (\varphi_i = Msg) \wedge \exists j \in \{1, \cdots, N\}, \exists M \in \Sigma^* : \mathcal{F}_{ji} = (x, val_x) \cdot M$$

- $s' = s[\mathcal{F}_{ji} \mapsto M]$
  This transition reads a message of the form (x, val_x) that was sent by process j. It will process its assignment using the rules defined in section 3.3.

Read execution

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{S}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i = Msg) \wedge \exists j \in \{1, \cdots, N\}, \exists M \in \Sigma^* : \mathcal{F}_{ji} = S \cdot M$

- $s' = s[\mathcal{F}_{ji} \mapsto M]$
  This transition reads an execution (caused by a launch) that came from process j. It will process this execution using the rules defined in section 3.3.

Read sequence

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i = Msg) \wedge \exists S \in \mathcal{M}_i$

- $s' = s[\mathcal{M}_i \mapsto \mathcal{M}_i \backslash \{S\}; \mathcal{A}_i \mapsto \mathcal{A}_i \cup \{S\}]$
  This transition takes a distributed execution that was sent by another process, and puts it in the active set. It will get executed in the processing phase.

End of message reading

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i = Msg)$

- $s' = s[\varphi_i \mapsto Processing]$
  At any time, process i can stop reading messages and pass on to the processing phase.

3.3 Processing transitions

Assignment

$$< \cdots \| \overbrace{\texttt{x := expr;}S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{W_1; \cdots W_k}^{i} \| \cdots >$$
if $s \models (\varphi_i \in \{Processing, Msg\}) \wedge (Var(expr) \cup \{x\} \subseteq Var(Expr_i))$

- $s' = s_1[\mathcal{V}(i)[\![x]\!] \mapsto \mathcal{V}(i)[\![expr]\!]; \mathcal{A}_i = \mathcal{A}_i \cup \{S_2\}; \forall j \in \{1, \cdots, N\} \backslash \{i\} : \mathcal{F}_{ij} \mapsto \mathcal{F}_{ij} \cdot (x, \mathcal{V}(i)[\![x]\!])]$
  The first part evaluates the expr, and updates the variable with the new value, the second part puts the rest of the instructions in the waiting set, and finally, the third part sends the new value to all other processes.
- $1, \cdots, k : W_k$ are the whens associated to process i
- $s_1 = s[\mathcal{A}_i \mapsto \mathcal{A}_i \cup \mathcal{S}_i; \mathcal{S}_i \mapsto \phi]$
  Some of the sleeping sequences may have their wait condition go true. Make them awaiting execution.

Assignment$_d$

$$< \cdots \| \overbrace{\texttt{x := expr;}S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i \in \{Processing, Msg\}) \wedge (Var(expr) \cup \{x\} \subseteq Var(Expr_j) \wedge j \neq i)$

- $s' = s[\mathcal{M}_j \mapsto \mathcal{M}_j \cup \{\texttt{x := expr;}S_2\}]$
  Since process i can't evaluate the expression or do the assignment, the sequence is sent over to the process that can.

If$_1$

$$< \cdots \| \overbrace{\textbf{if expr then S end_if } S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{S; S_2}^{i} \| \cdots, s >$$
if $s \models (\varphi_i \in \{Processing, Msg\} \wedge \mathcal{V}(i)[\![expr]\!] = \top) \wedge (expr \in Expr_i)$

If$_0$

$$< \cdots \| \overbrace{\textbf{if expr then S end_if } S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{S_2}^{i} \| \cdots, s >$$
if $s \models (\varphi_i \in \{Processing, Msg\} \wedge \mathcal{V}(i)[\![expr]\!] = \bot) \wedge (expr \in Expr_i)$

If$_d$

$$< \cdots \| \overbrace{\textbf{if expr then S end_if } S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i = Processing) \wedge (expr \in Expr_j \wedge j \neq i)$

- $s' = s[\mathcal{M}_j \mapsto \mathcal{M}_j \cup \{\textbf{if expr then S end_if } S_2\}]$
  Since process i can't evaluate the expression, the sequence is sent over to the process that can evaluate it.

While

$$< \cdots \| \overbrace{\textbf{while expr do S end_while } S_2}^{i} \| \cdots, s >$$
$$\downarrow \tau \qquad \text{if } s \models (\varphi_i \in \{Processing, Msg\}) \wedge (expr \in Expr_i)$$
$$< \cdots \| \overbrace{\textbf{if expr then S; while expr do S end_while } S_2}^{i} \| \cdots, s >$$

While$_d$

$$< \cdots \| \overbrace{\textbf{while expr do S end_while } S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i = Processing) \wedge (expr \in Expr_j \wedge j \neq i)$

- $s' = s[\mathcal{M}_j \mapsto \mathcal{M}_j \cup \{\textbf{while expr do S end_while } S_2\}]$
  Since process i can't evaluate the expression, the sequence is sent over to the process that can evaluate it.

Wait$_1$

$$< \cdots \| \overbrace{\textbf{wait expr; } S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{S_2}^{i} \| \cdots, s >$$
if $s \models (\varphi_i = Processing \wedge \mathcal{V}(i)[\![expr]\!] = \top) \wedge (expr \in Expr_i)$

Wait$_0$

$$< \cdots \| \overbrace{\textbf{wait expr; } S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i = Processing \wedge \mathcal{V}(i)[\![expr]\!] = \bot) \wedge (expr \in Expr_i)$

- $s' = s[\mathcal{S}_i \mapsto \mathcal{S}_i \cup \{\text{wait expr; } S_2\}]$

Wait$_d$

$$< \cdots \| \overbrace{\texttt{wait expr; } S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' >$$
if $s \models (\varphi_i = Processing) \wedge (expr \in Expr_j \wedge j \neq i)$

- $s' = s[\mathcal{M}_j \mapsto \mathcal{M}_j \cup \{\texttt{wait expr}; S_2\}]$
  Since process i can't evaluate the expression, the sequence is sent over to the process that can evaluate it.

Launch

$$< \cdots \| \overbrace{\texttt{Launch id;} S_2}^{i} \| \cdots, s > \xrightarrow{\tau} < S_2, s' >$$
if $s \models (\varphi_i \in \{Processing, Msg\})$

- $s' = [\mathcal{F}_{ij} \mapsto \mathcal{F}_{ij} \cdot \mathcal{L}(id)]$
  Let j be the process that can execute the instructions associated to id. Define $\mathcal{L}(id)$ as the function that maps an id to the statement list associated to that id.

End Of Process$_{1,r}$

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' > \quad \text{if } \mathcal{A}_i = \phi \wedge s \models (\varphi_i = Processing)$$

- $s' = s[\varphi_i \mapsto Msg]$

End Of Process$_{1,o}$

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' > \quad \text{if } \mathcal{A}_i = \phi \wedge s \models (\varphi_i = Processing)$$

- $s' = s[\varphi_i \mapsto Output]$

End Of Process$_0$

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{\tau} < \cdots \| \overbrace{S}^{i} \| \cdots, s' > \quad \text{if } \exists S \in \mathcal{A}_i \wedge s \models (\varphi_i = Processing)$$

- $s' = s[\mathcal{A}_i \mapsto \mathcal{A}_i \backslash \{S\}]$

3.4 Output

Output

$$< \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s > \xrightarrow{!\{f:OVar_i \mapsto \{\top,\bot\}\}} < \cdots \| \overbrace{\epsilon}^{i} \| \cdots, s' > \quad \text{if } s \models (\varphi_i = Output)$$

- $s' = s[\varphi_i \mapsto Input]$
  After performing the output, pass to the input phase

- $f : OVar_i \mapsto \{\top, \bot\} = x \mapsto \mathcal{V}(i)[\![x]\!]$
  This function specifies the new values for the output variables

Annexe C

Manuel d'utilisation LegOS

D.1. Capteur de lumière

LIGHT_*X* : variable contenant la valeur lue par le capteur de lumière situé en position *X* (égal à 1, 2 ou 3).

ds_passive(*SENSOR*_X) : fonction de mise en mode passif du capteur situé en position *X* (1, 2 ou 3).

ds_active(*SENSOR*_X) : fonction de mise en mode actif du capteur situé en position *X* (1, 2 ou 3).

D.2. Capteur de toucher

TOUCH_*X* : variable contenant l'état du capteur de toucher en position *X* (1, 2 ou 3). Renvoie 1 si le bouton est enfoncé, 0 sinon.

D.3. Bouton

PRESSED(button_state(), *NAME*) : macro permettant de tester si le bouton *NAME* (BUTTON_RUN ou BUTTON_VIEWED) est enfoncé.

RELEASED(button_state(), *NAME*) : macro permettant de tester si le bouton *NAME* (BUTTON_RUN ou BUTTON_VIEWED) est relâché.

D.4. Moteur

motor_*X*_dir(enum *motor_dir*) : fonction permettant de mettre à jour la direction d'un moteur situé en position *X* (A, B ou C). ***motor_dir*** peut prendre les valeurs **fwd** (avant), **rev** (arrière), **off** (arrêt) ou **brake** (roues bloquées).

motor_*X*_speed(int motor_speed) : fonction permettant de mettre à jour la vitesse d'un moteur

situé en position *X* (A, B ou C). ***motor_speed*** doit être compris entre MIN_SPEED (0) et MAX_SPEED(255).

D.5. Écran LCD

cputw(int *i*) : fonction permettant d'afficher l'entier *i* sur l'écran.

cputs(char* *s*) : fonction permettant d'afficher le string *s* (de longueur maximale de 5 caractères) sur l'écran.

lcd_clear() : fonction permettant d'effacer l'écran.

D.6. Multithreading

execi(&*function*, int argc, char **argv, int prior, DEFAULT_STACK_SIZE) : fonction permettant de lancer une tâche exécutant les instructions contenue dans *function*. Les paramètres **argc** et **argv** contiennent respectivement le nombre et les paramètres de la tâche. La priorité (**prior**) est comprise entre PRIO_LOWEST(0) et PRIO_HIGHEST(20). La fonction renvoie l'identificateur de la tâche (**pid**).

kill(int *pid*) : fonction qui permet de tâche la ta^che identifiée par *pid*.

sleep(int *s*) : fonction permettant de mettre en attente une tâche pendant *s* secondes.

msleep(int *m*) : fonction permettant de mettre en attente une tâche pendant *m* millisecondes.

wait_event(&*function*, wakeup_t *data*) : fonction permettant de suspendre une tâche en attendant que *function* renvoie une valeur non nulle. *data* permet de passer des paramètres à la fonction.

D.7. Lego Network Protocol

lnp_init(int *tcp_host*, int *tcp_port*, int *lnp_adress*, int *lnp_mask*, int *flags*) : fonction permettant d'initialiser le protocole. Pour chaque paramètre, 0 est la valeur par défaut. *tcp_host* et *tcp_port* sont à utiliser dans un programme qui s'exécutera sur un terminal pour se connecter au LNP daemon (*lnpd*). *lnp_adress* et *lnp_mask* permettent de spécifier l'adresse LNP du programme. *Flags* permet de modifier le comportement de lnpd.

lnp_adressing_set_handler(int *port*, *portHandler_function*) : fonction permettant de spécifier un gestionnaire du port *port*. Lorsqu'un aquet est reçu, le système d'exploitation appelle *portHandler_function* pour le traiter. Les paquets sont transmis de manière adressée.

lnp_adressing_write(unsigned char* *data*, unsigned char *length*, int *dest_addr*, int *port*) : fonction permettant d'envoyer de manière adressée le message *data* de longueur *length* à l'adresse *dest_addr*, par l'intermédiairie du port *port*.

lnp_integrity_set_handler(int *port*, *portHandler_function*) : fonction permettant de spécifier un gestionnaire du port *port*. Lorsqu'un aquet est reçu, le système d'exploitation appelle

portHandler_function pour le traiter. Les paquets sont transmis de manière broadcastée.

lnp_integrity_write(unsigned char* *data*, unsigned char *length*, int *port*) : fonction permettant d'envoyer de manière broadcastée le message *data* de longueur *length* par l'intermédiairie du port *port*.

Annexe D

Code du fichier dsl_vm.h

```
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>

/*INCLUDE de LEGOS*/
#include <dbutton.h>
#include <dsensor.h>
#include <dmotor.h>
#include <lnp/lnp.h>
#include <dsound.h>
#include <sys/tm.h>

#include <conio.h>
#include <unistd.h>
#include <semaphore.h>

#include <lnp/sys/irq.h>
#define EC() disable_irqs()
#define XC() enable_irqs()


#define ABS(a) ((a) > 0 ? (a) : (-a))
#define halt() while(1);

#define HOP_WAIT_ID 65200
#define ERR_UNKNOWN_WARN_ID   1
#define ERR_UNKNOWN_EXEC_ID   2
#define ERR_UNKNOWN_HOP_ID    3
#define ERR_BUFFER_EMPTY      4
#define ERR_BUFFER_OVERLOAD   5
#define ERR_OUT_OF_MEMORY     6
#define ERR_STACK_EMPTY       7
#define ERR_UNKNOWN_SEQ_ID    8
#define ERR_WRITE_FAIL        9
#define ERR_UNKNOWN_MSG_ID   10
#define ERR_BUFFER_OVERLOAD2 11

#define DS_ALL_ACTIVE //sensors activation
#define true  1
#define false 0
#define BOOL  unsigned char
#define INT   short
#define DINT  short
#define ULINT time_t
#define SIZE_MAX 8

#define ab_write ab_awrite
```

```
#define init_AB_protocol(a) init_ab_protocol(portHandler, ID_SITE, 4, 4, 4)

#define INT_TO_CHAR(s, i) *(s) = *((char*) &i); *(s+1) = *(((char*) &i)+1)
#define CHAR_TO_INT(i, s) *((char*) &i) = *(s); *(((char*) &i)+1) = *(s+1);
#define ADD_INT_TO_CHAR(s,i) *(s++) = *((char*) &i); *(s++) = *(((char*) &i)+1)
#define READ_INT_FROM_CHAR(i,s) *((char*)&i)= *(s++); *(((char*) &i)+1) = *(s++)
#define READ_DINT_FROM_CHAR(i,s) READ_INT_FROM_CHAR(i,s)
#define READ_ULINT_FROM_CHAR(i,s) READ_INT_FROM_CHAR(i,s)
#define READ_BOOL_FROM_CHAR(i,s) READ_INT_FROM_CHAR(i,s) //bool := 1 or 0

pid_t getID(int s);

char * dsl_vm_buf[SIZE_MAX];
pid_t* dsl_vm_stack[SIZE_MAX];

int POS_BUF=0, ST_BUF=0;

int POS_STACK, SIZE_STACK, ST_STACK;

int dsl_vm_id;
char* dsl_vm_args;

time_t builtin_cms;

BOOL _time_setup;

unsigned int _id_send_message=0;

sem_t sem_port_handler, sem_SIZE_BUF;
pid_t* current_thread_id;


/*functions declarations*/
int startSequences();
void handleWarn(int id, int val);
void handleExec(int id, unsigned char * args);
void handleWhens();
void sampleInputs();
void writeOutputs();
void handleMsg();
void timer();

int start() {
  int i = 0;
  while(true) {
    builtin_cms = sys_time;
    sampleInputs();
    handleWhens();
    handleMsg();
    writeOutputs();
  }
}

void error(int i) {
  cputs("error");
  cputw(i);
  dsound_system(0);
  sleep(1);
}

/*read input*/
int sample(int i) {
  switch(i){
  case 1: return SENSOR_1;
    break;
  case 2: return SENSOR_2;
```

```
    break;
  case 3: return SENSOR_3;
  }
  return 0;
}


wakeup_t getControl(wakeup_t pid)
{
  return ((unsigned int)current_thread_id == pid);
}

/*data manipulation*/
void pushStack(pid_t* pid)
{
  if(SIZE_STACK >= SIZE_MAX) {
    error(ERR_OUT_OF_MEMORY);
    halt();
  }
  else {
    dsl_vm_stack[SIZE_STACK] = pid;
    SIZE_STACK = SIZE_STACK + 1;
  }
}

pid_t* popStack()
{
  if(SIZE_STACK <= 0) {
    error(ERR_STACK_EMPTY);
    halt();
  }
  else {
    SIZE_STACK = SIZE_STACK - 1;
    return dsl_vm_stack[SIZE_STACK];
  }
}

#define StackIsEmpty() (SIZE_STACK==0)

void startwait() {
  if (!StackIsEmpty()) {
    current_thread_id = popStack();
  }
  wait_event(&getControl,(unsigned int)cpid);
}

void wakeup(pid_t * id) {
  pushStack((pid_t*)cpid);
  current_thread_id = id;
  wait_event(&getControl, (unsigned int)cpid);
}

void portHandler(const unsigned char *message, unsigned char length){
  int SIZE_BUF;
  unsigned char * ptr;

  sem_getvalue(&sem_SIZE_BUF,&SIZE_BUF);
  if(SIZE_BUF>=SIZE_MAX) {
    error (ERR_BUFFER_OVERLOAD);
  } else {
    if(!(ptr=(char*) malloc(length))) {
      error (ERR_BUFFER_OVERLOAD2);
    }  else {
      memcpy(ptr, message, length);
      dsl_vm_buf[POS_BUF] = ptr;
      POS_BUF = (POS_BUF+1) % SIZE_MAX;
```

```
      sem_post(&sem_SIZE_BUF);
    }
  }
}

void systime_setup_handler(const unsigned char *message, unsigned char length){
  time_t time;
  CHAR_TO_INT(time,message);
  sys_time = time;
  _time_setup = true;
}

void systime_send(){
  char message[2];
  time_t time=sys_time;
  INT_TO_CHAR(message,time);
  lnp_integrity_write(message, 2);
}

void systime_setup() {
  int temp=0;
  _time_setup=false;
  lnp_integrity_set_handler(systime_setup_handler);
  while(!_time_setup) {
    temp++; cputw(temp);
  }
}

void send (unsigned char* message, unsigned char length,unsigned destination) {
  if(!ab_awrite(message, length, destination)) error (ERR_WRITE_FAIL);
}

void LAUNCH_ID_COLOR(int site, int id, int param, ...) {
  const unsigned char ln = 3 + 2*param;
  char message[ln], *p = message;
  va_list ap;
  int i;

  va_start(ap, param);

  *(p++)=2;
  ADD_INT_TO_CHAR(p, id);
  while (param) { //none id transmitted
    i = va_arg(ap, int);
    ADD_INT_TO_CHAR(p, i);
    param--;
  }

  send(message, ln, site);
  va_end(ap);
}

void SEND_ID_WARN (int site,int id, int val){
  char message[5], *p = message;
  *(p++)=3;
  ADD_INT_TO_CHAR(p,id);
  ADD_INT_TO_CHAR(p,val);
  send(message, 5, site);
}

void HOP(int site, int sequence, int id, int n_para, ...) {
  const unsigned char ln = (n_para*2)+6;
  char message[ln], *p = message;
  unsigned char i;
  int arg;
```

```
  va_list ap;
  va_start(ap,n_para);

  *(p++)=1;
  ADD_INT_TO_CHAR(p,sequence);
  ADD_INT_TO_CHAR(p,id);
  *(p++)=n_para;

  for (i = 0; i <n_para*2; i++) {//For each parameters:[id,val] are transmitted
    arg = va_arg(ap, int);
    ADD_INT_TO_CHAR(p, arg);
  }
  send(message, ln, site);
  va_end(ap);
}

#define SELF_HOP(sequence, id) HOP(ID_SITE, sequence, id, 0)

void handleMsg() {
  int nbVal, i, nMax, _id, val, seq;
  unsigned char * ptr, * type;
  pid_t * seq_id;

  sem_getvalue(&sem_SIZE_BUF,&nMax);

  while (sem_trywait(&sem_SIZE_BUF) == 0 && nMax--) {
    type = ptr = dsl_vm_buf[ST_BUF];
    ptr++;
    if((*type) == 1) //launch sequence
      {
        READ_INT_FROM_CHAR(seq,ptr);
        READ_INT_FROM_CHAR(dsl_vm_id,ptr);
        if(!(dsl_vm_args=(char*) malloc(((*ptr)*2)+1))) {
          error(ERR_OUT_OF_MEMORY);
          halt();
        } else{
          dsl_vm_args=ptr;
          pushStack((pid_t*)cpid);
          seq_id = (pid_t*) getID(seq);
          if(seq_id) {
            current_thread_id = seq_id;
            wait_event(&getControl,(unsigned int) cpid);
          } else {
            error (ERR_UNKNOWN_SEQ_ID);
          }
        }
      } else {
        READ_INT_FROM_CHAR(_id,ptr);
        if(*type == 2) {//launch method
          handleExec(_id,ptr);
        } else if (*type == 3) {//tild
          READ_INT_FROM_CHAR(val,ptr);
          handleWarn(_id, val);
        } else {
          error (ERR_UNKNOWN_MSG_ID);
        }
    }
    ST_BUF = (ST_BUF+1) % SIZE_MAX;
    free(type);
  }
}

void init(){
  POS_BUF=0, ST_BUF=0;
  POS_STACK=0, SIZE_STACK=0, ST_STACK=0;
  dsl_vm_id=0;
```

```
  current_thread_id = 0;

  sem_init(&sem_SIZE_BUF,1,0);
}

#if 1
wakeup_t button_press_wakeup(wakeup_t data)
{
   return (PRESSED(dbutton(), data));
}
#endif
```

Annexe E

Code du protocole de communication

ab.h

```
/*
  Implementation of the alternating bit protocol on top of LNP for
  LegOS/BrickOS
  Copyright (C) 2004  Bram De Wachter

  This program is free software; you can redistribute it and/or
  modify it under the terms of the GNU General Public License
  as published by the Free Software Foundation; either version 2
  of the License, or (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*/


/*! \mainpage Alternating Bit Protocol
 *
 * \section contents Package Contents
 *
 *
 * This package implements the Alternating Bit Protocal (AB-Protocol)
 * in C with legOS/BrickOS system calls.
 *
 * \htmlonly
  For more information on the alternating bit protocol, look
      <a
href="http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?Alternating+bit+protocol">her
e</A>
      [FOLDOC]
  \endhtmlonly
 *
 * The AB-protocol makes the communciations between Lego-RCXs reliable.
 * The importance of a reliable protocol is undeniable since the standard
 * communication primitives with LegOS/BrickOS, based on LNP guarantee only
 * the integrity of packets, <B>not</B> their delivery.
 *
 * This protocol implementation does not achive miracles : two bricks that are
 * definately out of range will never communicate (unless using a nifty
```

```
 * ad hoc protocol). The protocol will continue to try to send the message,
 * and can notify the sender when the message was received.
 *
 * This package is multi-threaded, robust, efficient and small
 * (in that order).
 *
 * \section interface The AB-Protocols interface
 *
 * Three functions control the AB-Protocol :
 * 1. The init_ab_protocol, which inits all data structures, 2. ab_swrite
 * for synchronous message sending (the send function returns when the message
 * is delivered) and 3. ab_awrite for asynchronous message sending (the send
 * function returns immediately).
 *
 * Look at the ab protocol interface module documentation for more details.
 *
 * We used this implementation in highly asynchronous and multithreaded
 * environment, and took special care to avoid race-conditions, this
 * implementation should therefore be sufficiently robust to survive
 * the more demanding developer.
 *
 * \section license License
 *
 * This software is released under the Gnu Public License (GPL).
 * Feel free.
 *
 * \section Compiling
 *
 * To compile the library, you must have legOS/BrickOS 0.2.6 (probably
 * works with other versions too) and the h8300 cross-compiler (which is
 * also needed to use legOS/BrickOS) installed.
 *
 * Next, change LEGOS_ROOT in the Makefile in src/lib so it points
 * to your legOS installation directory.
 *
 * Launch <b>Make</b> to create the <b>ablib.a</b> library.
 *
 * Any legos program should then <b>#include "ab.h"</b> (located in
 * src/include)
 * and the <b>ablib.a</b> file  must be linked with the program.
 *
 * Look at the example program.
 *
 * \section contacts Contacts/Version
 *
 * \author Bram De Wachter (<B>bdewacht</B> <I>a-t</I> <B>ulb</B>
 * <I>d-o-t</I> <B>ac</B> <I>d-o-t</I> <B>be</B>)
 * \author  Nicolas Devos
 * \date    April, 2004
 * \version 1.0
 */

/*Alternating Bit Protocol:
  1. use init_AB_protocol(portHandler) to initialize the protocol
  2. use AB_write(message, length) to send a message
*/

#ifndef AB_H__
#define AB_H__
#include <lnp/lnp.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <semaphore.h>

/** \defgroup ab_interface Ab protocol interface
```

```
 * \brief These functions should be called to get the AB protocol up
 * and running
 */
//@{

/// Set to 1 to get some debugging on LCD
#define DEBUG     0

/** \brief Set to 1 if pools are to be used instead of malloc,
    Setting to 0 may cause problems if more memory is asked than available
*/
#define AB_STATIC 0

//@}

/** \defgroup ab_internals Ab protocol internals
 * Internal function for the AB protocol
 */
//@{

/** \brief Structure used to store incoming and outgoing messages
 */
typedef struct mess {
  /// The pointer to the actual message contents
  unsigned char* MSG;

  /// The message length,
  /// limited to 255 characters
  unsigned char LENGTH;

  /// For use with synchronous message sending,
  /// this semaphore will be posted when the message is acknowledged
  sem_t         sem_ack;

  /// Linked list: points to the next element
  struct mess *NEXT;
} MESS ;


/** \brief Structure used to store other communicating sites
 */
typedef struct site {
  /// The distant site's id, 255 different ids supported
  unsigned char ID;

  /// Send and reception bit, used in the alternating bit protocol
  /// bit0 : send
  /// bit1 : recv
  unsigned char RSBIT;

  /// Linked list of sites
  struct site *NEXT;

  /// Pending messages (outgoing) for this site
  struct mess *LIST;
} SITE ;

#if AB_STATIC
/**
   \struct pool_element
   \brief A pool_element points to a pre-allocated chunk of memory.

   Basic implementation, uses a linked list of pool_elements,
   each of which can be free or taken
*/
typedef struct pool_element {
```

```
  /// The pre-allocated memory zone
  void                * cont;

  /// The next zone
  struct pool_element * next;

  /// Indicates if this chunk is free or not
  char                  free;
} pool_el;

/**
   \brief A pool is a set of pre-allocated chunks of memory.

   Basic implementation, uses a pointer to the first allocated chunk
   of memory, and a semaphore that indicated the number of free elements
*/
typedef struct {
  /// Semaphore indicates the number of free elements
  sem_t     size;

  /// The first element
  pool_el * head;
} pool;
#endif
//@}

/** \addtogroup ab_interface */
//@{

/**  \brief Use this function to asynchronously send a message.

    The message is added to the list and will be sent as soon as possible. The
function returns
    immediately
    \param  message The message to send
    \param  length The length of the message
    \param  destination The destination for the message
    \return A pointer to the MESS structure if successfully added to the list,
NULL otherwise
*/
MESS * ab_awrite(const unsigned char* message, unsigned char length, unsigned
char destination);

/**  \brief Use this function to synchronously send a message.

    The message is added to the list and will be sent as soon as possible. The
function returns
    when an acknowledgment is received for the message
    \param  message The message to send
    \param  length The length of the message
    \param  destination The destination for the message
    \return A pointer to the MESS structure if success, NULL otherwise
*/
MESS * ab_swrite(const unsigned char* message, unsigned char length, unsigned
char destination);

/**
   \brief Initialises everything needed for use of reliable protocol

   \param u_portHandler The function that will be called when a message is
received
   \param id_site       The id of the current site
   \param n_sites       The number participating distant sites
   \param incoming      The number of simultaneously incoming messages
   \param outgoing      The number of simultaneously outgoing messages
   \return 0 on success 1, otherwise (memory allocation problems)
```

```
   \warning When AB_STATIC is not used, the parameters n_sites, incoming and
ougoing are ignored
*/
int init_ab_protocol(void (*u_portHandler) (const unsigned char *message,
unsigned char length),unsigned char id_site, unsigned char n_sites, unsigned
char incoming, unsigned char outgoing);

//@}
//

#endif
```

ab.c

```
#include "ab.h"

/*
  Implementation of the alternating bit protocol on top of LNP for
  LegOS/BrickOS
  Copyright (C) 2004  Bram De Wachter

  This program is free software; you can redistribute it and/or
  modify it under the terms of the GNU General Public License
  as published by the Free Software Foundation; either version 2
  of the License, or (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*/

/** \addtogroup ab_internals */
//@{

#if DEBUG
#define DEBUG_AB(debug_instr)  debug_instr
#else
#define DEBUG_AB(debug_instr)
#endif

#define TRUE                        1
#define FALSE                       0

#define INCOMING                    0
#define OUTGOING                    1

#define ERR_AB_WRITE               10
#define DEFAULT_HEADER_SIZE         3
#define MSG_HEAD(msg)          msg[0]
#define MSG_SOURCE(msg)        msg[1]
#define MSG_DEST(msg)          msg[2]

#include <lnp/sys/irq.h>
#define ENTER_CRITICAL() disable_irqs()
#define EXIT_CRITICAL()  enable_irqs()

#if AB_STATIC
/// Preallocated memory for incoming messages
pool  IMESS_POOL;
```

```
/// Preallocated memory for outgoing messages
pool  OMESS_POOL;

/// Preallocated memory for sites
pool SITE_POOL;
#endif

/// Semaphores for mutual exclusion and producer/consumer
sem_t sem_ab_write,sem_ab_portHandler,sem_todo;

/// list of current messages to (re)send
SITE* S_FIRST;

/// list of received messages
MESS* M_FIRST;

/// Current site's id
unsigned char _id_site;

/// Pointer to the user's porthandler (for incoming messages, dispatched to the
user)
void (*user_portHandler) (const unsigned char *buf, unsigned char len);

/*
   Message :

      0          1           2      3 ... payload ... N
   MSG_HEAD  MSG_SOURCE  MSG_DEST

   MSG_HEAD :
   7 6 5 4 3 2 1 0  (8 bits)
  HOB           LOB

   0 : bit for abp
   1 : is_ack

   => HEAD = 0 : message with bit = 0
             1 : message with bit = 1
             2 : ack with bit     = 0
           3 : ack with bit     = 1
*/
#define set_ack(msg,val)   MSG_HEAD(msg) = (val ? (MSG_HEAD(msg)|0x2) :
(MSG_HEAD(msg)&0xFD))
#define set_ack_1(msg)     MSG_HEAD(msg) |= 0x2;
#define set_ack_0(msg)     MSG_HEAD(msg) &= 0xFD;
#define is_ack(msg)         (MSG_HEAD(msg)&0x2)

#define set_bit(msg,val)   MSG_HEAD(msg) = (val ? (MSG_HEAD(msg)|0x1) :
(MSG_HEAD(msg)&0xFE))
#define set_bit_1(msg)     MSG_HEAD(msg) |= 0x1;
#define set_bit_0(msg)     MSG_HEAD(msg) &= 0xFE;
#define get_bit(msg)        (MSG_HEAD(msg)&0x1)


#if AB_STATIC
/**************************************************************************/
/* Memory pool  functions                                                 */
/**************************************************************************/


/*!
  \brief Initialises a pool.

  This function populates a pool with chunks of memory of the same size
  using malloc()
```

```
  \param p The pool
  \param n The number of elements
  \param s The size of each element
  \return 0 if successfull, 1 if not enough memory
*/
char init_pool(pool * p, int n, int s) {
  pool_el * pe;
  sem_init(&p->size,1,n);
  for (p->head = NULL; n >= 0; n--) {
    if((pe = malloc(s+sizeof(pool_el))) == NULL) return 1;
    pe->next = p->head;
    pe->free = 1;
    pe->cont = ((char*)pe)+sizeof(pool_el);
    p->head = pe;
  }
  return 0;
}

/*!
  \brief Gets the first free element, and marks it as 'not free'

  \param head The pool's first element
  \return the content pointed to by the first free element, NULL if none
  \warning disables IRQs to make concurrent access safe.
  Linear in the size of the pool
*/
void * pool_get_first(pool_el * head) {
  ENTER_CRITICAL();
  for(; head; head=head->next) if(head->free) { head->free = FALSE; break; }
  EXIT_CRITICAL();
  return head ? head->cont : NULL;
}

/*!
  \brief Marks an element as 'free' in a pool.

  \param head The pool's first element
  \param el   The chunk of allocated memory
  \warning disables IRQs to make concurrent access safe.
  Linear in the size of the pool
*/
void pool_free(pool_el * head, void * el) {
  ENTER_CRITICAL();
  for(; head; head=head->next) if(head->cont == el) { head->free = TRUE;
break; }
  EXIT_CRITICAL();
}


/*!
  \brief Allocates a chunk from the pool, and waits until an element is freed
  in case no elements are free.

  \param    p The pool
  \return   A pointer to the first free element
*/
void * wait_alloc_pool(pool * p)          { sem_wait(&p->size); return
pool_get_first(p->head); }


/*!
  \brief Allocates a chunk from the pool.

  \param    p The pool
  \return   A pointer to the first free element, NULL if none where free
*/
```

```
void *      alloc_pool(pool * p)          { return sem_trywait(&p->size) == 0 ?
pool_get_first(p->head) : NULL; }

/*!
  \brief De-allocates a chunk in the pool
  \param p   The pool
  \param el  The contetns to be freed
*/
void         free_pool(pool * p,void *el) { pool_free(p->head,el); sem_post(&p-
>size); }

#define      alloc_imess(l)   ((MESS*)      alloc_pool(&IMESS_POOL))
#define wait_alloc_imess(l)   ((MESS*) wait_alloc_pool(&IMESS_POOL))
#define       free_imess(m)                  free_pool(&IMESS_POOL,m)
#define      alloc_omess(l)   ((MESS*)      alloc_pool(&OMESS_POOL))
#define wait_alloc_omess(l)   ((MESS*) wait_alloc_pool(&OMESS_POOL))
#define       free_omess(m)                  free_pool(&OMESS_POOL,m)
#define       alloc_site()   ((SITE*)      alloc_pool(&SITE_POOL))
#else
/*****************************************************************************/
/** DYNAMIC ALLOCATION                                                       **/
/*****************************************************************************/

inline SITE * alloc_site() {
  return malloc(sizeof(SITE));
}
inline MESS * alloc_mess(unsigned char length) {
  return malloc(sizeof(MESS)+DEFAULT_HEADER_SIZE+length);
}
inline void free_omess(MESS * mess) {
  free (mess);
}
inline void free_imess(MESS * mess) {
  free (mess);
}
#endif



/*****************************************************************************/
/** AB PROTOCOL                                                              **/
/*****************************************************************************/

#define flip_sb(rs)    rs^=0x1
#define flip_rb(rs)    rs^=0x2
#define get_sb(rs)     (rs&0x1)
#define get_rb(rs)     ((rs&0x2)>>1)


/*!
  \brief Erases the first message for a given site.
  \param site The site
*/
static void erase(SITE * site ) {
  MESS * msg;
  msg = site->LIST;
  if (msg) {
    site->LIST=msg->NEXT;
    free_omess(msg);
  }
}

/*! \brief Creates a new message.
  \param header The header for the new message see definitions of MSG_xxxx
  \param msg    The message
  \param length The length of msg
```

```
  \param wait   If 1, will block until memory is freeµ
  \param io     Used to take from pool INCOMING or OUTGOING
  \warning if AB_STATIC is not used, the wait parameter is ignored, since we
have no
  control on memory allocation
  \return a new message, or NULL if memory was not available and wait is false
*/
static MESS * new_msg(const unsigned char * header, const unsigned char* msg,
unsigned char length,
                  unsigned char wait, unsigned char io){
  MESS *m;

#if AB_STATIC
  if (wait) {
    if (io == INCOMING) m = wait_alloc_imess(length);
    else m = wait_alloc_omess(length);
  } else {
    if (io == INCOMING) m = alloc_imess(length);
    else m = alloc_omess(length);
  }
#else
  m = alloc_mess(length);
#endif
  if (!m) return NULL;

  // Combine the two mallocs to only one, make m->MSG point to the area after
the MESS structure
  m->MSG = ((char*)m)+sizeof(MESS);
  m->LENGTH=length + DEFAULT_HEADER_SIZE;
  sem_init(&m->sem_ack, 1, 0);
  memcpy(m->MSG, header, DEFAULT_HEADER_SIZE);
  memcpy(m->MSG+DEFAULT_HEADER_SIZE, msg, length);

  m->NEXT=NULL;

  return m;
}

/** \brief Creates a new site
    \param id_site The new site's id
    \return a fresh SITE or NULL if no more memory available
 */
static SITE * new_site(unsigned char id_site){
  SITE *s;
  if ((s = alloc_site()) == NULL ) return NULL;
  s->ID_SITE=id_site;
  s->LIST=0;
  s->RSBIT=0;
  return s;
}

/**
   Adds a message to the outgoing messages to a certain site
   \param site The destination site
   \param m    The message
*/
static void add( SITE *site, MESS *m ) {
  MESS *i=site->LIST;
  if (m) {
    if(!i) site->LIST=m;
    else {
      while(i->NEXT) i=i->NEXT;
      i->NEXT=m;
    }
  }
}
```

```
/**
   \brief Searches for a site with a given id, and creates one if not found
   \param   id_site The id to look for
   \return  The site with the given id, NULL if not found and no more available
memory
*/
static SITE * search_and_add(unsigned char id_site) {
  SITE* i;
  for (i=S_FIRST; i; i=i->NEXT) if (i->ID_SITE == id_site) break;
  if (!i) {
    i = S_FIRST;
    S_FIRST = new_site(id_site);
    if (S_FIRST) S_FIRST->NEXT = i;
    i = S_FIRST;
  }
  return i;
}

/**
   \brief This is the ISR for the infrared.

   Incoming messages are put into the linked list pointed to by M_FIRST, and
will be treated
   by one of the protocol's threads
*/
static void AB_portHandler(const unsigned char *message, unsigned char length){
  MESS *curr;

  if(length>=DEFAULT_HEADER_SIZE){
    if(MSG_DEST(message) == _id_site) {
      if(sem_trywait(&sem_ab_portHandler)==0) {/*list not in use in thread AB*/
      if ((curr = new_msg(message,
                        message+DEFAULT_HEADER_SIZE,
                        length-DEFAULT_HEADER_SIZE,0,INCOMING)) != NULL) {
        curr->NEXT = M_FIRST;

  M_FIRST    = curr;
        sem_post(&sem_todo);
      }
      sem_post(&sem_ab_portHandler);
      }
    }
  }
}

/**
   \brief Thread handling incoming messages
*/
static int thread_AB_I () {
  SITE* S;
  MESS* M, *t;
  unsigned char* _MSG;
  char ack[DEFAULT_HEADER_SIZE];

  while(1) {

    sem_wait(&sem_todo);
    sem_post(&sem_todo);

    sem_wait(&sem_ab_write);

    //treatment of received messages
    sem_wait(&sem_ab_portHandler);
    M=M_FIRST;
    M_FIRST=NULL;
```

```
    sem_post(&sem_ab_portHandler);

    while(M != NULL) {
      _MSG = M->MSG;
      sem_trywait(&sem_todo);

      if(is_ack(_MSG)) { // ACK RECEIVED
      DEBUG_AB(cputs("ack"));

      //    sem_wait(&sem_ab_write);
      S=search_and_add(MSG_SOURCE(_MSG));
      if(S) {
        if(get_sb(S->RSBIT)==get_bit(_MSG) ) {
          sem_post(&S->LIST->sem_ack);
          erase(S);
          flip_sb(S->RSBIT);
        }
      }
      // sem_post(&sem_ab_write);
      } else {    // Message received,
      // answer with an ack containing same bit
      MSG_SOURCE(ack) = MSG_DEST  (_MSG);
      MSG_DEST  (ack) = MSG_SOURCE(_MSG);
      MSG_HEAD  (ack) = 0;
      set_ack_1 (ack);
      set_bit   (ack,get_bit(_MSG));

      // Send ack
      lnp_integrity_write(ack, DEFAULT_HEADER_SIZE);
      msleep(0);

      // Find the site that sent this message
      // sem_wait(&sem_ab_write);
      S=search_and_add(MSG_SOURCE(_MSG));

      // Site found, if bit corresponds, call user
      if(S) {
        if(get_rb(S->RSBIT) == get_bit(_MSG)) {
          user_portHandler(_MSG+DEFAULT_HEADER_SIZE,
                       M->LENGTH-DEFAULT_HEADER_SIZE);
          flip_rb(S->RSBIT); //switch BIT of site
        }
      }
      // sem_post(&sem_ab_write);
      }

      // Message treated, destroy it
      t=M;
      M=M->NEXT;
      free_imess(t);
    } // Loop on messages

    sem_post(&sem_ab_write);

  } // Loop while(1)
}

/**
   \brief Thread handling outgoing messages
*/
static int thread_AB_O () {
  SITE* S;
  unsigned char* _MSG;

  while(1) {
    msleep (250);
```

```
    sem_wait(&sem_ab_write);
    S=S_FIRST;
    while(S) {
      if(S->LIST!=NULL) {
      _MSG=S->LIST->MSG;
      if (MSG_DEST(_MSG) == _id_site) {
        user_portHandler(_MSG+DEFAULT_HEADER_SIZE,
                     S->LIST->LENGTH-DEFAULT_HEADER_SIZE);
        erase (S);
      } else {
        MSG_HEAD(_MSG)=0;
        set_bit(_MSG,get_sb(S->RSBIT));
        lnp_integrity_write(_MSG,S->LIST->LENGTH);
        msleep(0);
      }
      }
      S=S->NEXT;
    }
    sem_post(&sem_ab_write);
  }
  return 0;
}
//@}


/** \addtogroup ab_interface */
//@{

/**
   \brief Initialises everything needed for use of reliable protocol
   \param u_portHandler The function that will be called when a message is
received
   \param id_site       The id of the current site
   \param n_sites       The number participating distant sites
   \param incoming      The number of simultaneously incoming messages
   \param outgoing      The number of simultaneously outgoing messages
   \return 0 on success 1, otherwise (memory allocation problems)
   \warning When AB_STATIC is not used, the parameters n_sites, incoming and
ougoing are ignored
*/
int init_ab_protocol(void (*u_portHandler) (const unsigned char *message,
unsigned char length),  unsigned char id_site, unsigned char n_sites,
                 unsigned char incoming, unsigned char outgoing) {
#if AB_STATIC
  if (init_pool(&IMESS_POOL, incoming, sizeof(MESS)+DEFAULT_HEADER_SIZE+255) ||
      init_pool(&OMESS_POOL, outgoing, sizeof(MESS)+DEFAULT_HEADER_SIZE+255) ||
      init_pool(&SITE_POOL, n_sites, sizeof(SITE))) {
    return 1;
  }
#endif

  S_FIRST=NULL;
  M_FIRST=NULL;
  _id_site = id_site;
  sem_init(&sem_ab_portHandler,1,1);
  sem_init(&sem_ab_write,1,1);
  sem_init(&sem_todo,1,0);
  user_portHandler = u_portHandler;
  lnp_integrity_set_handler(AB_portHandler);

  execi(&thread_AB_I, 0,0,PRIO_NORMAL+1, DEFAULT_STACK_SIZE);
  execi(&thread_AB_O, 0,0,PRIO_NORMAL+1, DEFAULT_STACK_SIZE);

  return 0;
}
```

```
/**  \brief Use this function to asynchronously send a message.

    The message is added to the list and will be sent as soon as possible. The
function returns
    immediately
    \param  message The message to send
    \param  length The length of the message
    \param  destination The destination for the message
    \return A pointer to the MESS structure if successfully added to the list,
NULL otherwise
*/
MESS * ab_awrite(const unsigned char* message, unsigned char length, unsigned
char destination) {
  char msg_header[DEFAULT_HEADER_SIZE];
  SITE* site;
  MESS* mess = NULL;

  MSG_HEAD(msg_header)   = 0;
  MSG_SOURCE(msg_header) = _id_site;
  MSG_DEST(msg_header)   = destination;

  sem_wait(&sem_ab_write);
  site=search_and_add(destination);
  sem_post(&sem_ab_write);

  if (site) {
    mess = new_msg(msg_header, message, length, 0, OUTGOING);
    if (mess)  {
      sem_wait(&sem_ab_write);
      add(site, mess);
      sem_post(&sem_ab_write);
    }
  }

  return mess;
}
/**  \brief Use this function to synchronously send a message.

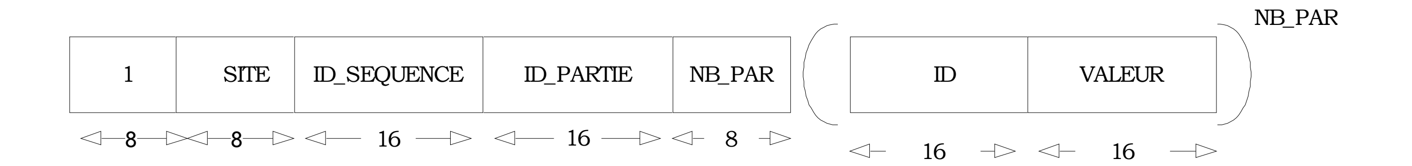
    The message is added to the list and will be sent as soon as possible. The
function returns
    when an acknowledgment is received for the message
    \param  message The message to send
    \param  length The length of the message
    \param  destination The destination for the message
    \return A pointer to the MESS structure if success, NULL otherwise
*/
MESS * ab_swrite(const unsigned char* message, unsigned char length,
             unsigned char destination) {
  MESS * mess = ab_awrite(message, length, destination);
  if (mess) sem_wait(&mess->sem_ack);
  return mess;
}

//@}
```

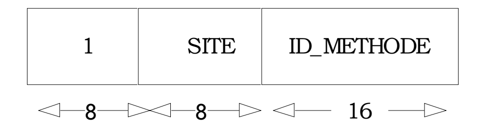
Annexe F

Format des 3 types de messages échangés

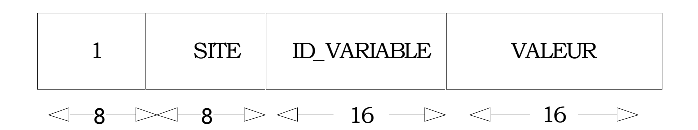
Message HOP



Message LAUNCH



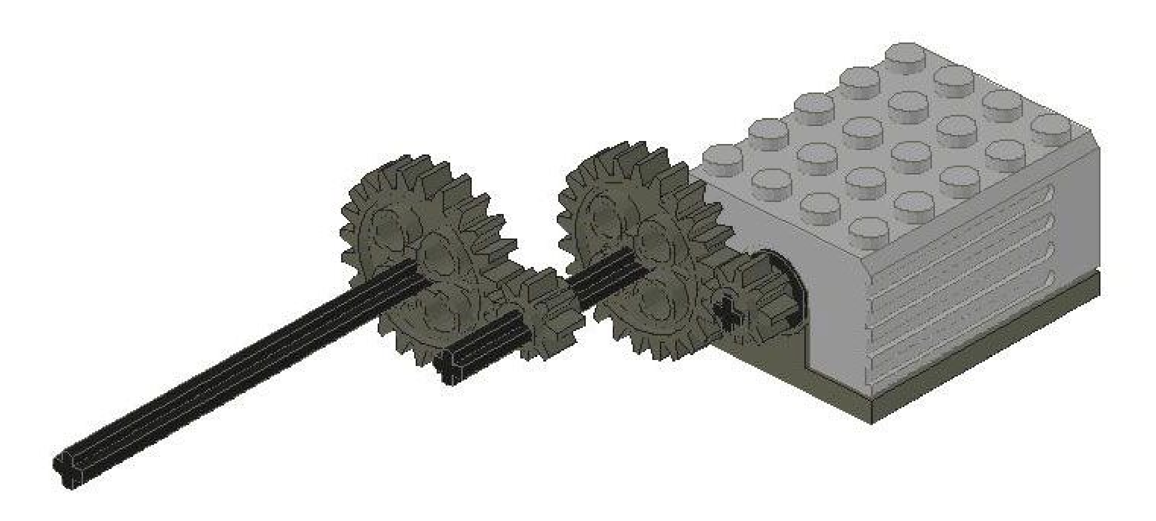
Message TILD



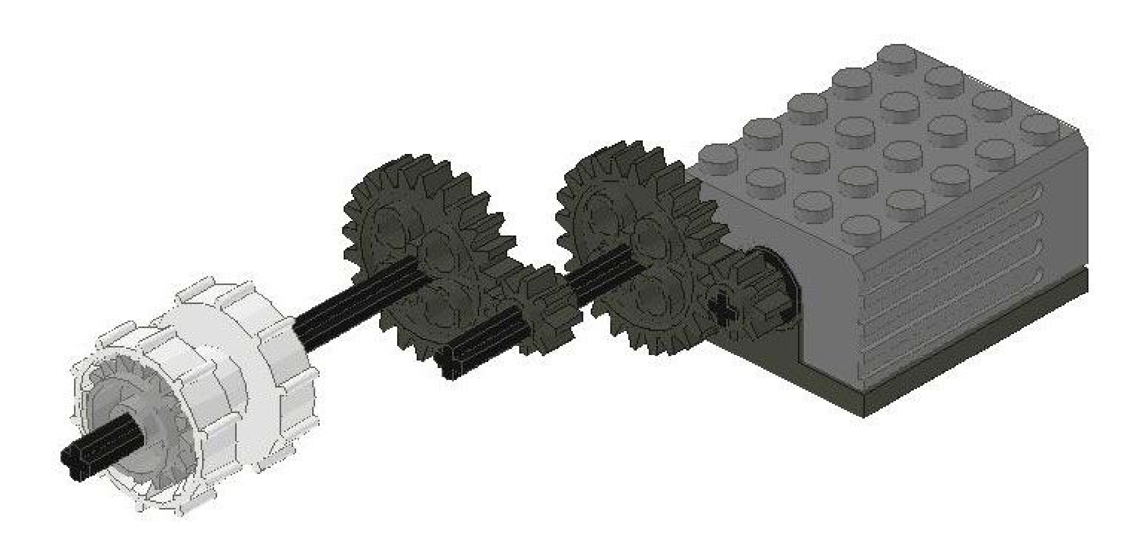
Annexe G

Principe de construction des tapis roulants en Lego-Mindstorms

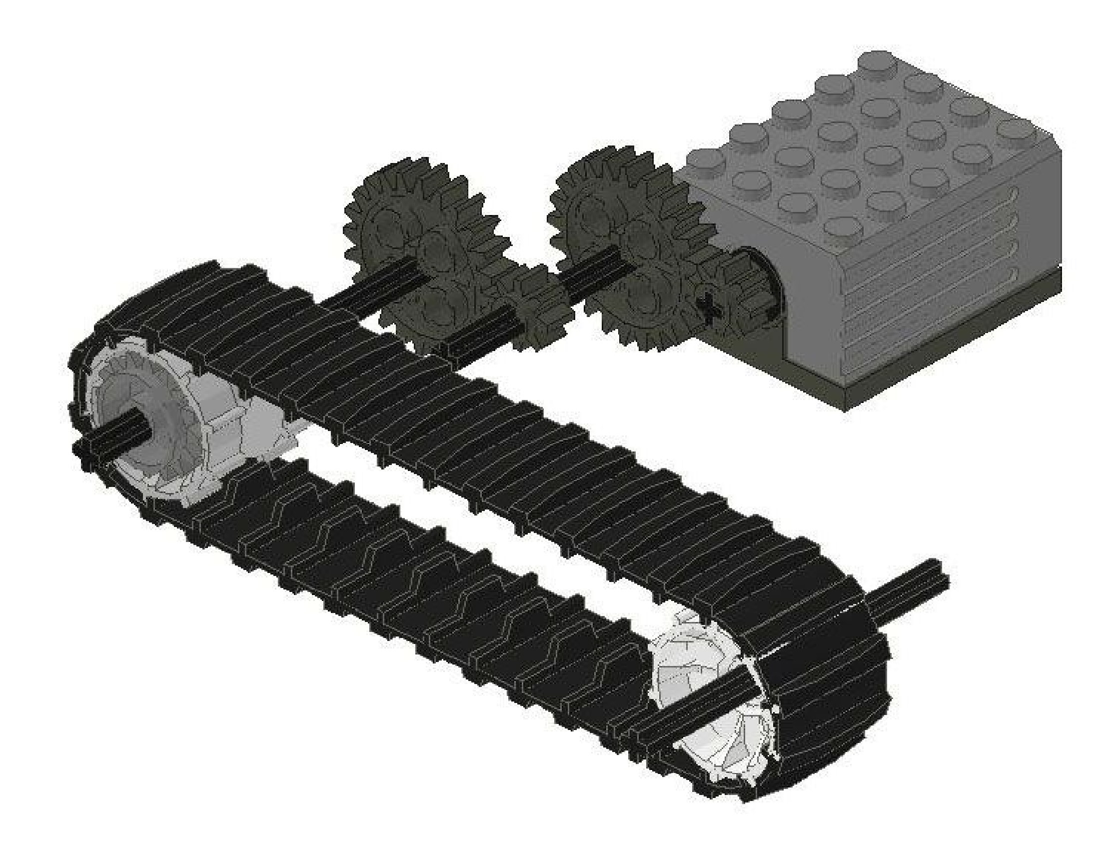
Etape 1



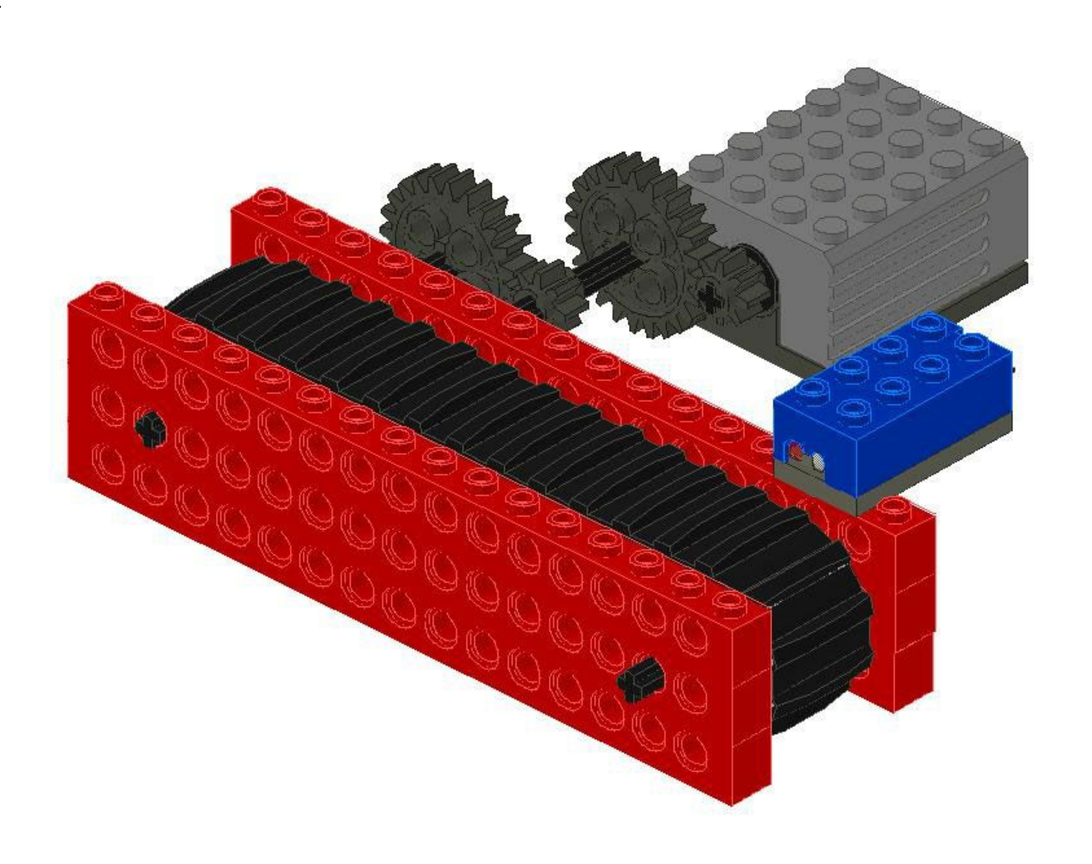
Etape 2



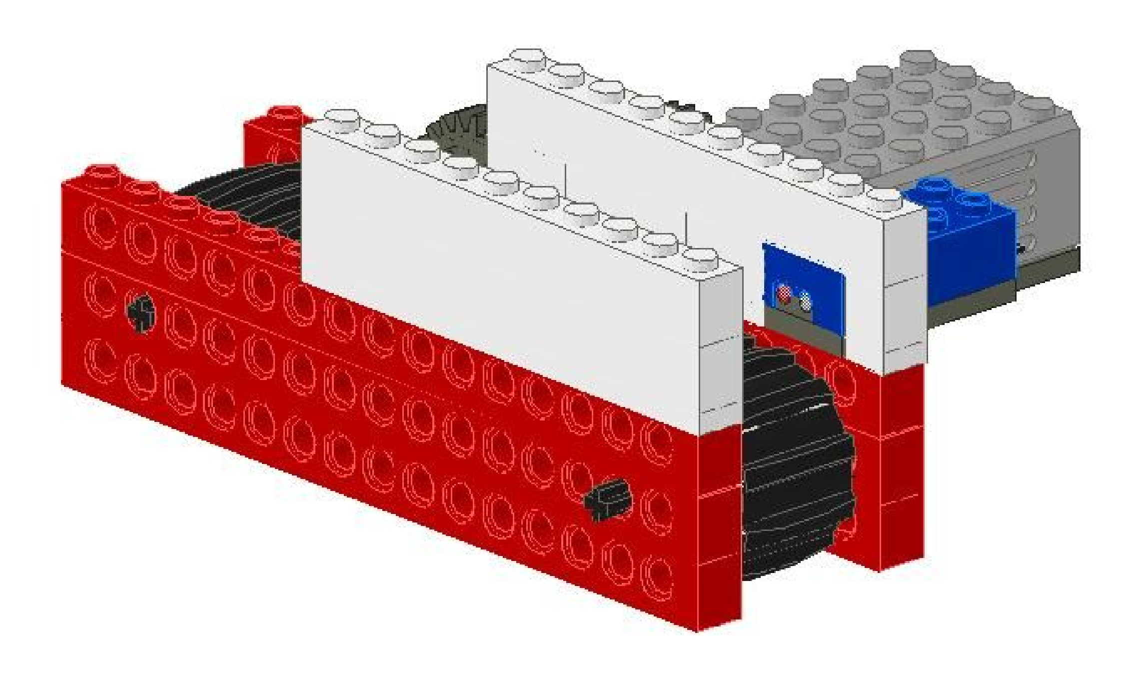
Etape 3



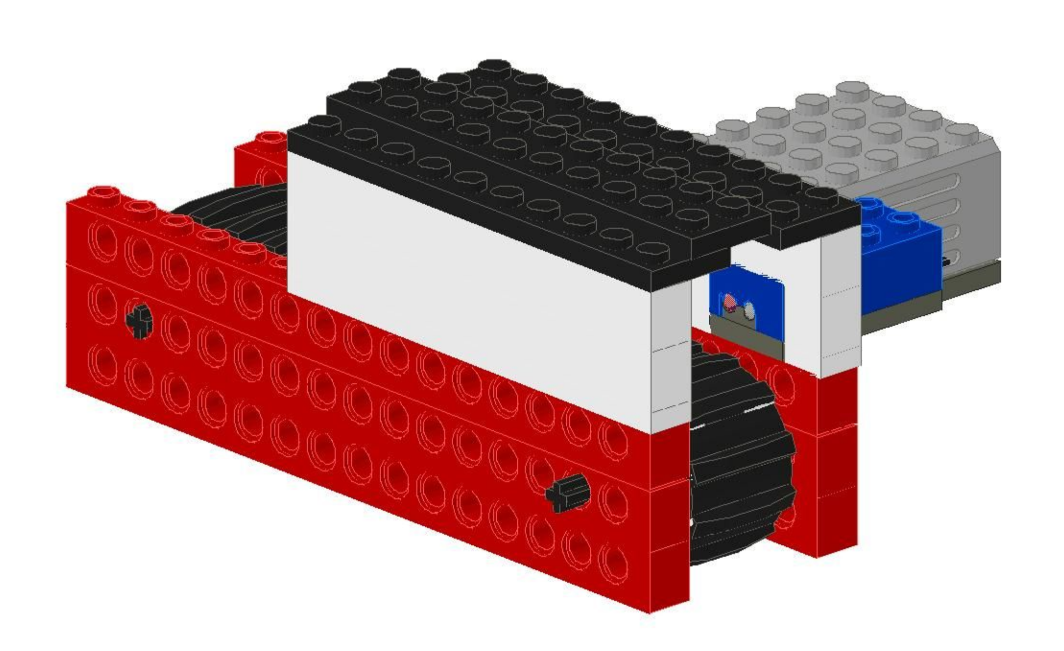
Etape 4



Etape 5



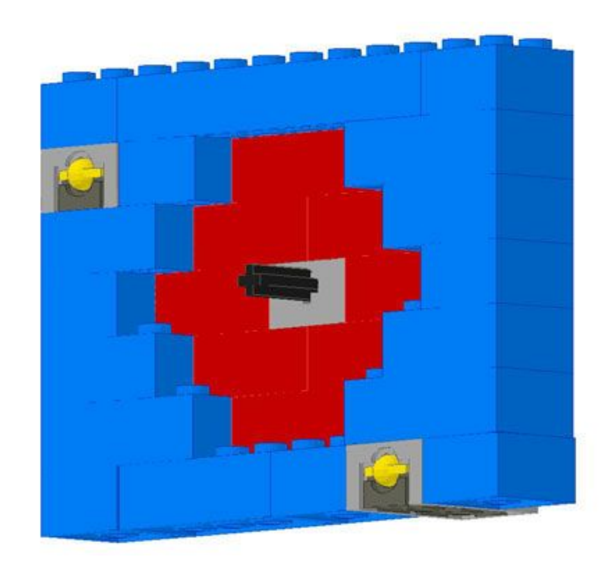
Etape 6



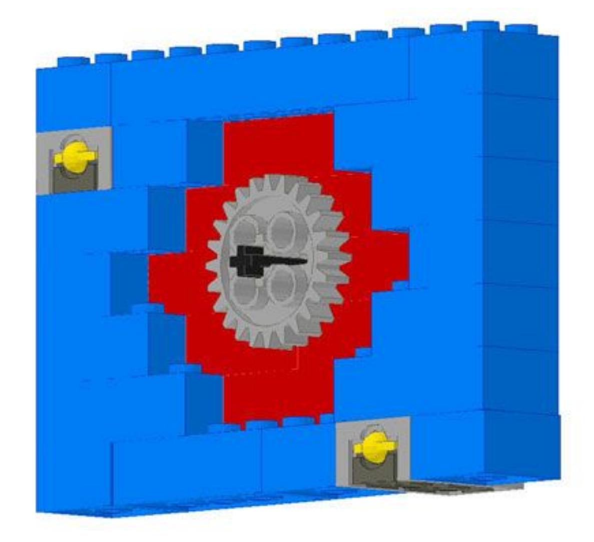
Annexe H

Principe de construction des chariots mobiles en Lego-Mindstorms

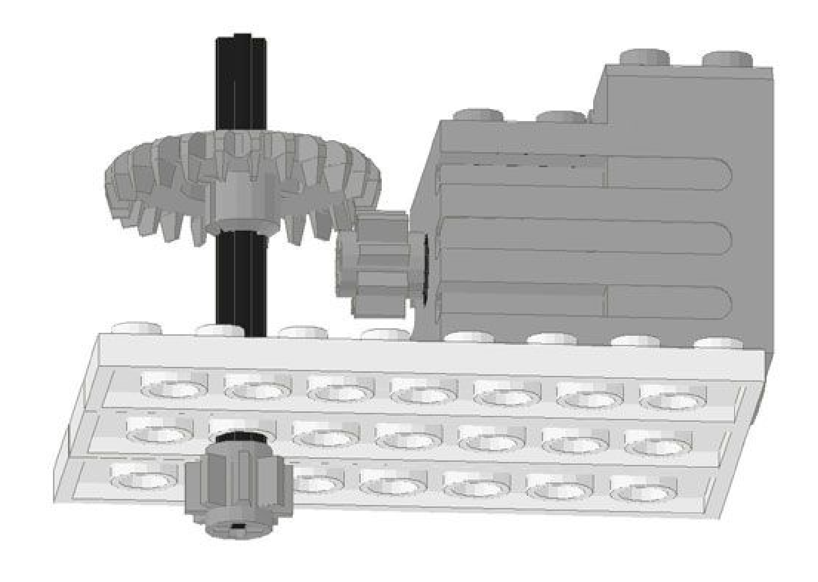
Etape 1



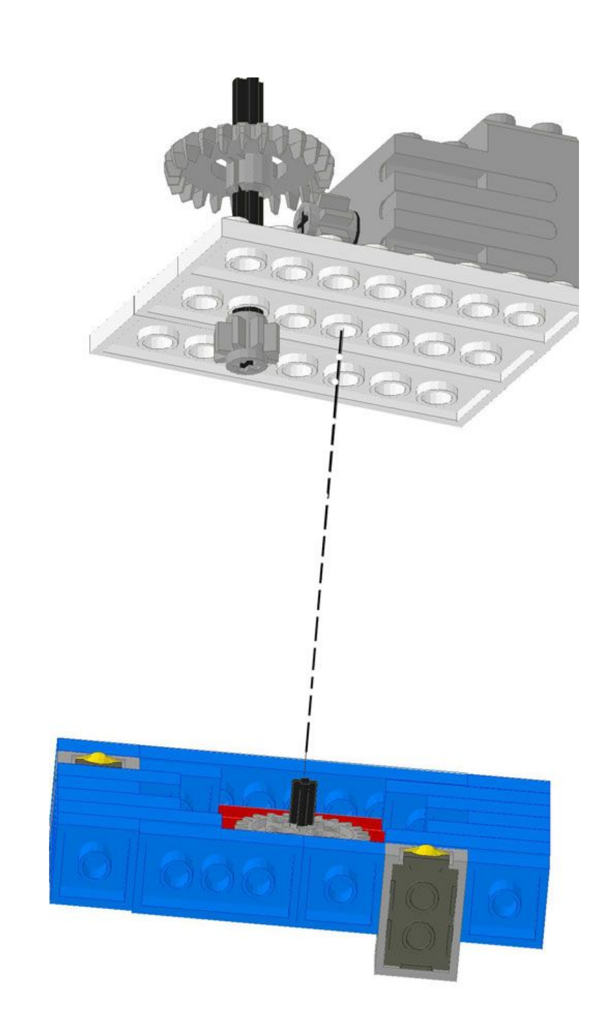
Etape 2



Etape 3

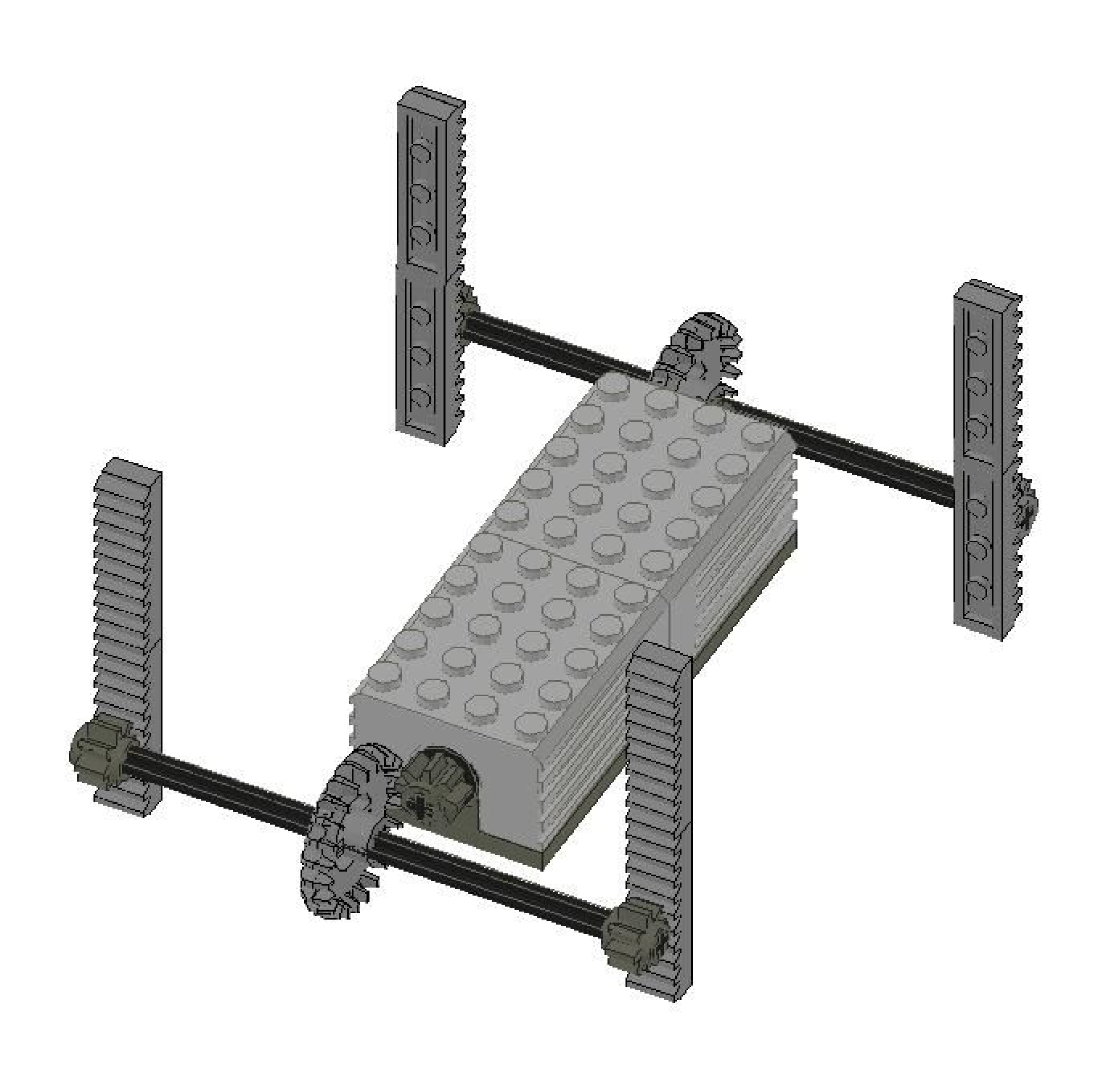


Etape 4



Annexe I

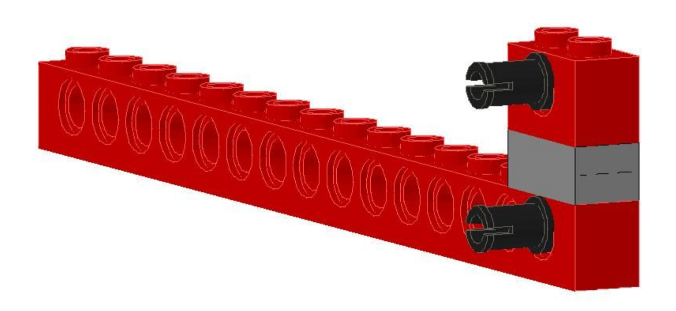
Principe de construction des ascenceurs en Lego-Mindstorms



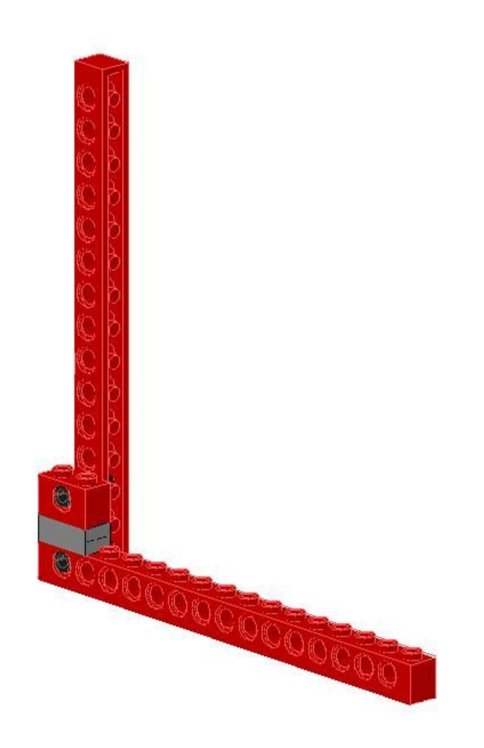
Annexe J

Principe de construction de la cage d'ascenceur en Lego-Mindstorms

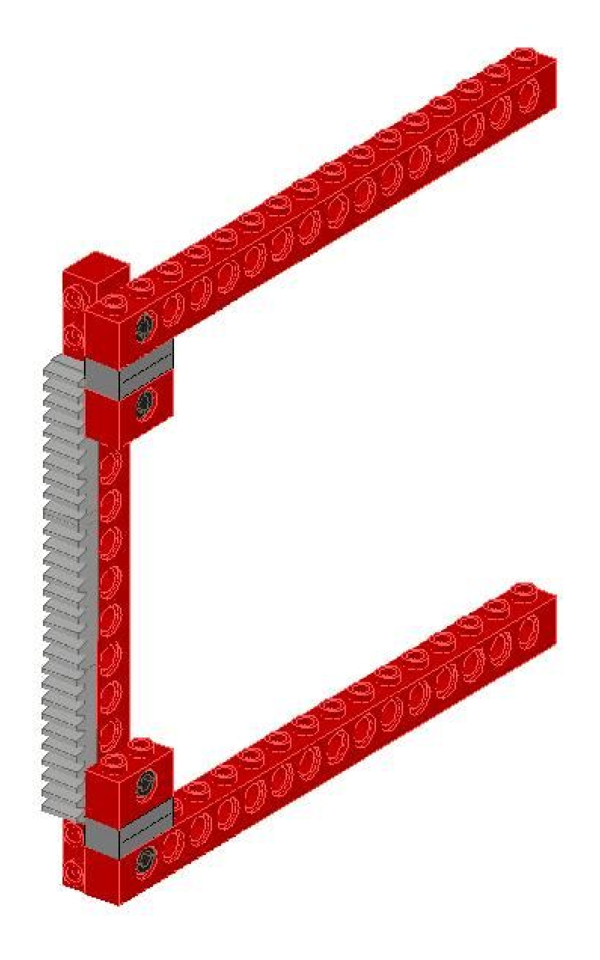
Etape 1



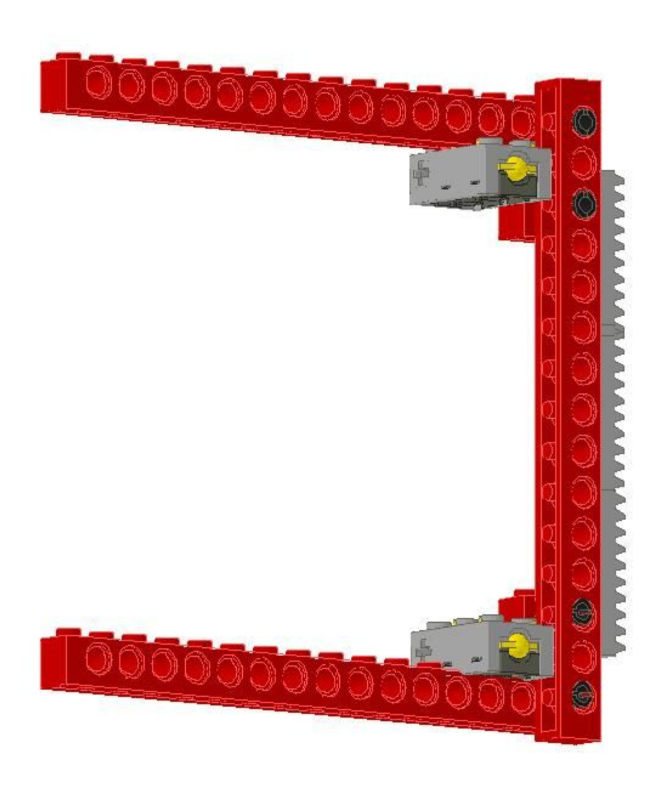
Etape 2



Etape 3



Etape 4



Annexe K

Code dSL de la chaîne de montage

```
#define TRESHOLD 39 (*treshold value of light sensors*)

(***********************
 *******CLASSES*********
 **********************)

CLASS conveyor_belt
   motor : INT;
   light : LONG;   (*light sensor*)
   free : BOOL;    (*state*)
END_CLASS

CLASS moving_cart
   motor : INT;
   touch_one : INT;      (*touch sensor position 1*)
   touch_two : INT;      (*touch sensor position 2*)
   rotor : conveyor_belt;
   position : BOOL;      (*TRUE:position 1, FALSE:position 2*)
   state : BOOL;         (*TRUE:running, FALSE:stopped*)
   free : BOOL;
END_CLASS

CLASS elevator
   motor : INT;
   touch_up : INT;        (*touch sensor position up*)
   touch_down : INT;      (*touch sensor position down*)
   rotor : conveyor_belt;
   position : BOOL;       (*TRUE:up, FALSE:down*)
   state : BOOL;          (*TRUE:running, FALSE:stopped*)
   free : BOOL;
END_CLASS

(************************
 ******DISTRIBUTION******
 ***********************)

GLOBAL_VAR
   t_1, t_2, t_3, t_4, t_5, t_6 : conveyor_belt;
   c_1, c_2 : moving_cart;
   a_1 : elevator;
   ask_one, ask_two : BOOL;
   last_in, last_out : BOOL;
END_VAR

SITE s1
   INPUT t_1.light : 0.0.0 ;
   INPUT t_2.light : 0.1.0 ;
```

```
   OUTPUT t_1.motor : 1.0.1 ;
   OUTPUT t_2.motor : 1.1.1 ;
END_SITE

SITE s2
   INPUT c_1.touch_one : 0.0.1;
   INPUT c_1.touch_two : 0.1.1;
   INPUT c_1.rotor.light : 0.2.0;
   OUTPUT c_1.motor : 1.0.1;
   OUTPUT c_1.rotor.motor : 1.2.1;
END_SITE

SITE s3
   INPUT t_3.light : 0.0.0 ;
   INPUT t_4.light : 0.1.0 ;
   OUTPUT t_3.motor : 1.0.1 ;
   OUTPUT t_4.motor : 1.1.1 ;
END_SITE

SITE s4
   INPUT a_1.touch_down : 0.0.1;
   INPUT a_1.touch_up : 0.1.1;
   INPUT a_1.rotor.light : 0.2.0;
   OUTPUT a_1.motor : 1.0.1;
   OUTPUT a_1.rotor.motor : 1.1.1;
END_SITE

SITE s5
   INPUT c_2.touch_one : 0.0.1;
   INPUT c_2.touch_two : 0.1.1;
   INPUT c_2.rotor.light : 0.2.0;
   OUTPUT c_2.motor : 1.0.1;
   OUTPUT c_2.rotor.motor : 1.2.1;
END_SITE

SITE s6
   INPUT t_5.light : 0.0.0 ;
   INPUT t_6.light : 0.1.0 ;
   OUTPUT t_5.motor : 1.0.1 ;
   OUTPUT t_6.motor : 1.1.1 ;
END_SITE

(*****************************
 ****CLASS FUNCTIONALITIES****
 *****************************)

(********conveyor_belt********)

METHOD conveyor_belt::go()
   self.motor := 20;
END_METHOD

METHOD conveyor_belt::stop()
   self.motor := 0;
END_METHOD

(********moving_cart********)

METHOD moving_cart::go_in_one()
   IF(self.touch_one<0) THEN
      self.state := FALSE;
      self.rotor<-stop();
      self.motor := -30;
   END_IF;
END_METHOD
```

```
METHOD moving_cart::go_in_two()
   IF(self.touch_two<0) THEN
      self.state := FALSE;
      self.rotor.motor:=0;
      self.motor := 30;
   END_IF;
END_METHOD

METHOD moving_cart::stop()
   self.motor := 0;
   self.state := TRUE;
END_METHOD

WHEN IN moving_cart self.touch_one > 0 THEN
   self.position := TRUE;
   self<-stop();
END_WHEN

WHEN IN moving_cart self.touch_two > 0 THEN
   self.position := FALSE;
   self<-stop();
END_WHEN

(********elevator********)

METHOD elevator::up()
   IF(self.touch_up<0) THEN
      self.state := FALSE;
      self.rotor<-stop();
      self.motor := 20;
   END_IF;
END_METHOD

METHOD elevator::down()
   IF(self.touch_down<0) THEN
      self.state := FALSE;
      self.rotor<-stop();
      self.motor := -2;
   END_IF;
END_METHOD

METHOD elevator::stop()
   self.motor := 0;
   self.state := TRUE;
END_METHOD

WHEN IN elevator self.touch_up > 0 THEN
   self.position := TRUE;
   self<-stop();
END_WHEN

WHEN IN elevator self.touch_down > 0 THEN
   self.position := FALSE;
   self<-stop();
END_WHEN

(************************
 ********SEQUENCES********
 ************************)

(***initialization***)
SEQUENCE MAIN()
   last_in:=FALSE;
   c_1<-go_in_one();
   c_1.free:=TRUE;
   t_3.free:=TRUE;
```

```
   a_1.free:=TRUE;
   c_2<-go_in_one();
   c_2.free:=TRUE;
   a_1<-down();
   last_out:=FALSE;
   t_1.motor:=20;
   t_2.motor:=20;
END_SEQUENCE

(***carry brick from t_4 to t_5***)
SEQUENCE t4_to_t5()

   (*get the moving cart 2 in position 1*)
   c_2<-go_in_one();
   wait(c_2.state==TRUE and c_2.position==TRUE);

   (*launch the two conveyer belts*)
   c_2.rotor<-go();
   LAUNCH t_4<-go();

   (*brick on conveyer belt of moving cart*)
   wait(c_2.rotor.light<TRESHOLD);
   LAUNCH t_4<-stop();
   c_2.rotor<-stop();

   (*wait the next conv. belt to be clear*)
   wait(t_5.light>=TRESHOLD);

   (*launch the two conveyer belts*)
   t_5<-go();
   LAUNCH c_2.rotor<-go();

   (*brick on conveyer belt 5*)
   wait(t_5.light<TRESHOLD);
   t_5<-stop();
   c_2.rotor<-stop();

   (*free the moving cart*)
   c_2.free:=TRUE;
END_SEQUENCE

(***carry brick from t_4 to t_6***)
SEQUENCE t4_to_t6()

   (*get the moving cart in position 1*)
   c_2<-go_in_one();
   wait(c_2.state==TRUE AND c_2.position==TRUE);

   (*launch the two conveyer belts*)
   c_2.rotor<-go();
   LAUNCH t_4<-go();

   (*brick on conveyer belt of moving cart*)
   wait(c_2.rotor.light<TRESHOLD);
   t_4<-stop();
   c_2.rotor<-stop();

   (*get the moving cart in position 2*)
   c_2<-go_in_two();
   wait(c_2.state==TRUE AND c_2.position==FALSE);

   (*wait the next conv. belt to be clear*)
   wait(t_6.light>=TRESHOLD);

   (*launch the two conveyer belts*)
   t_6<-go();
```

```
   LAUNCH c_2.rotor<-go();

   (*brick on conveyer belt 6*)
   wait(t_6.light<TRESHOLD);
   t_6<-stop();
   c_2.rotor<-stop();

   (*bring back the moving cart 2 in position 1*)
   LAUNCH c_2<-go_in_one();

   (*free the moving cart*)
   c_2.free:=TRUE;
END_SEQUENCE

(***carry brick from t_3 to t_4***)
SEQUENCE t3_to_t4()

   (*wait the elevator to be in position down*)
   a_1<-down();
   wait(a_1.position==FALSE AND a_1.state==TRUE);

   (*launch the two conveyer belts*)
   a_1.rotor<-go();
   t_3<-go();

   (*brick on conveyer belt of elevator*)
   wait(a_1.rotor.light<TRESHOLD);
   a_1.rotor<-stop();
   t_3<-stop();

   (*free the conveyer belt 3*)
   t_3.free:=TRUE;

   (*get the elevator in position up*)
   a_1<-up();
   wait(a_1.state==TRUE AND a_1.position==TRUE);

   (*launch the two conveyer belts*)
   t_4<-go();
   a_1.rotor<-go();
   wait(t_4.light<TRESHOLD);

   t_4<-stop();
   a_1.rotor<-stop();

   (*bring back the elevator in position down*)
   LAUNCH a_1<-down();

   (*wait the moving cart 2 to be free*)
   wait (c_2.free==TRUE);
   c_2.free:=FALSE;

   (*launch the next sequence*)
   IF (last_out==TRUE) THEN
      last_out:=FALSE;
      LAUNCH t4_to_t5();
   ELSE
      last_out:=TRUE;
      LAUNCH t4_to_t6();
   END_IF;

   (*free the elevator*)
   a_1.free:=TRUE;
END_SEQUENCE

(***carry brick from t_1 to t_3***)
```

```
SEQUENCE t1_to_t3()

   (*get the moving cart in position 1*)
   c_1<-go_in_one();
   wait(c_1.state==TRUE and c_1.position==TRUE);

   (*launch the two conveyer belts*)
   c_1.rotor<-go();
   LAUNCH t_1<-go();

   (*brick on conveyer belt of moving cart*)
   wait(c_1.rotor.light<TRESHOLD);
   c_1.rotor<-stop();

   (*wait the next conv. belt to be free*)
   wait(t_3.free==TRUE);
   t_3.free:=FALSE;

   (*launch the two conveyer belts*)
   t_3<-go();
   c_1.rotor<-go();

   (*brick on conveyer belt 3*)
   wait(t_3.light<TRESHOLD);
   t_3<-stop();
   c_1.rotor<-stop();

   (*wait the elevator to be free*)
   wait(a_1.free==TRUE);
   a_1.free:=FALSE;

   (*launch the next sequence*)
   LAUNCH t3_to_t4();

   (*free the moving cart*)
   c_1.free:=TRUE;
END_SEQUENCE

(***carry brick from t_2 to t_3***)
SEQUENCE t2_to_t3()
   (*get the moving cart in position 2*)
   c_1<-go_in_two();
   wait(c_1.state==TRUE AND c_1.position==FALSE);

   (*launch the two conveyer belts*)
   c_1.rotor<-go();
   LAUNCH t_2<-go();

   (*brick on conveyer belt of moving cart*)
   wait(c_1.rotor.light<TRESHOLD);

   (*get the moving cart in position 1*)
   LAUNCH c_1<-go_in_one();
   wait(c_1.state==TRUE AND c_1.position==TRUE);

   (*wait the next conv. belt to be free*)
   wait(t_3.free==TRUE);
   t_3.free:=FALSE;

   (*launch the two conveyer belts*)
   t_3<-go();
   LAUNCH c_1.rotor<-go();

   (*brick on conveyer belt 3*)
   wait(t_3.light<TRESHOLD);
   t_3<-stop();
```

```
   c_1.rotor<-stop();

   (*wait the elevator to be free*)
   wait(a_1.free==TRUE);
   a_1.free:=FALSE;

   (*launch the next sequence*)
   LAUNCH t3_to_t4();

   (*free the moving cart*)
   c_1.free:=TRUE;
END_SEQUENCE

(*************************
 *********WHENS*********
 *************************)
(*input conveyor belts always turning*)
WHEN t_1.light>=TRESHOLD THEN
   t_1<-go();
END_WHEN

WHEN t_2.light>=TRESHOLD THEN
   t_2<-go();
END_WHEN

(*****c_1 GESTION******)
WHEN ( t_2.light <TRESHOLD AND ask_two==FALSE) THEN
   t_2<-stop();
   ask_two:=TRUE;
END_WHEN

WHEN t_2.light >=TRESHOLD AND ask_two==TRUE THEN
   ask_two:=FALSE;
END_WHEN

WHEN (t_1.light < TRESHOLD) AND ask_one==FALSE THEN
   t_1<-stop();
   ask_one:=TRUE;
END_WHEN

WHEN t_1.light >=TRESHOLD AND ask_one==TRUE THEN
   ask_one:=FALSE;
END_WHEN

WHEN (ask_one==TRUE AND ask_two==FALSE AND c_1.free==TRUE) THEN
   c_1.free:=FALSE;    (*take control of the moving cart*)
   LAUNCH t1_to_t3();
   last_in:=TRUE;
END_WHEN

WHEN (ask_one==FALSE AND ask_two==TRUE AND c_1.free==TRUE) THEN
   c_1.free:=FALSE;    (*take control of the moving cart*)
   LAUNCH t2_to_t3();
   last_in:=FALSE;
END_WHEN

WHEN (ask_one==TRUE AND ask_two==TRUE AND c_1.free==TRUE) THEN
   c_1.free:=FALSE;    (*take control of the moving cart*)
   IF(last_in==FALSE) THEN
      LAUNCH t1_to_t3();
      last_in:=TRUE;
   ELSE
      LAUNCH t2_to_t3();
      last_in:=FALSE;
   END_IF;
END_WHEN
```